Karlsruher Institut für Technologie

KASTEL

Privacy-Preserving Collection and Analysis of User Data – Provably Secure and Practical

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Markus Felix Raiber

aus Mindelheim

Tag der mündlichen Prüfung: 16. April 2026
1. Referent/Referentin: Prof. Dr. Jörn Müller-Quade
2. Referent/Referentin: Prof. Markulf Kohlweiss, Ph.D.

Abstract

A large number of applications collect, store and analyze data about end users. Examples include customer loyalty systems such as Payback, incentive systems offered by health insurers, behavior-dependent car insurance tariffs, pay-as-you-go public transport services, smart metering, and many more besides. Currently, most of these systems simply collect raw user data, resulting in vast datasets of personal information. However, this has disadvantages for both users and companies. The collected data allows extensive profiles to be created that go far beyond the intended use. Large collections of data are also a lucrative target for cyber-attacks, which can harm affected users through identity theft, for example, as well as causing harm to the involved company through negative publicity and potential data protection penalties.

Privacy-preserving technologies are a remedy for these problems, as they allow the desired analytics to be evaluated securely on relevant user data without the need to collect or store sensitive user data in the clear. However, this comes with new challenges, as privacy-preserving technologies are more computationally and communicationally complex. In this thesis, we propose and evaluate two generic solutions based on these technologies. Both solutions are formally modelled in the Universal Composability framework, which allows them to be used in any context while maintaining strong security guarantees through simulation-based security. Furthermore, both solutions come with a practical prototype implementation and evaluation, showcasing the potential for practical deployment as well as the current limitations. In both solutions, we ensure that users remain anonymous when data is collected, while guaranteeing the authenticity of the collected data.

Our first solution, called *PUBA*, is based on personal logbooks stored on each user's device. These logbooks are authenticated and can only be updated by the system operator while maintaining the confidentiality of their content. Users can then participate in privacy-preserving analytics computation, where it is ensured that their logbook is up-to-date and authentic. To accommodate constrained user devices, such as smartphones, users can outsource more complex analytics computations to a (potentially malicious) proxy that is not colluding with the system operator. Performance evaluations of our prototype show that PUBA has sufficient performance for logbooks storing the last 10-30 transactions.

In our second solution, called *POBA*, the logbooks are stored on operator-controlled servers instead. We model a setting in which multiple operators collaborate to run the system without fully trusting each other. Logbook contents are protected by secret-sharing them between all the operators involved. Additionally, advanced cryptographic tools, such as oblivious RAM, are employed to protect user identities and prevent the linking of multiple interactions to the same user. Since data is available without user interaction

in this setting, operators have more flexibility when running analytics. As long as some operators behave honestly, requiring all operators to agree to computations still ensures that the analysis results satisfy privacy requirements. Performance evaluations of our prototype demonstrate its practicability in the three-party setting: With three operators, it can handle over two million logbook entries per day.

Zusammenfassung

Eine Vielzahl von Anwendungen sammelt, speichert und analysiert Daten über Endnutzer. Beispiele hierfür sind Kundenbindungssysteme wie Payback, Bonusprogramme von Krankenkassen, verhaltensabhängige Kfz-Versicherungstarife, Pay-as-you-go-Dienste im öffentlichen Nahverkehr, Smart Metering und viele andere. Derzeit werden von den meisten dieser Systeme Nutzerdaten in großem Umfang gesammelt, was zu umfangreichen Datensätzen mit personenbezogenen Informationen führt. Dies hat jedoch sowohl für Nutzer als auch für Unternehmen Nachteile. Die gesammelten Daten ermöglichen die Erstellung umfangreicher Profile, die weit über den beabsichtigten Verwendungszweck hinausgehen. Große Datensammlungen sind zudem ein lukratives Ziel für Cyberangriffe, die den betroffenen Nutzern beispielsweise durch Identitätsdiebstahl Schaden zufügen und den beteiligten Unternehmen durch negative Publicity und mögliche Datenschutzstrafen schaden können.

Technologien zum Schutz der Privatsphäre sind eine Lösung für diese Probleme, da sie es ermöglichen, die gewünschten Analysen sicher anhand relevanter Benutzerdaten durchzuführen, ohne dass sensible Daten im Klartext erfasst oder gespeichert werden müssen. Dies bringt jedoch neue Herausforderungen mit sich, da Technologien zum Schutz der Privatsphäre in Bezug auf Rechenleistung und Kommunikation anspruchsvoll sind. In dieser Arbeit schlagen wir zwei generische Lösungen vor, die auf diesen Technologien basieren, und bewerten sie. Beide Lösungen sind formal im Rahmen des Universal Composability Framework modelliert, wodurch sie in jedem Kontext eingesetzt werden können und dabei ihre durch simulationsbasierte Sicherheit hohen Sicherheitsgarantien behalten. Darüber hinaus werden beide Lösungen mit einer praktischen Prototyp-Implementierung und -Bewertung vorgestellt, die das Potenzial für den praktischen Einsatz sowie die aktuellen Einschränkungen aufzeigen. Bei beiden Lösungen stellen wir sicher, dass die Nutzer bei der Datenerfassung anonym bleiben und gleichzeitig die Authentizität der erfassten Daten gewährleistet ist.

Unsere erste Lösung namens *PUBA* basiert auf persönlichen Logbüchern, die auf dem Gerät jedes Benutzers gespeichert sind. Diese Logbücher sind authentifiziert und können nur vom Systembetreiber aktualisiert werden, wobei die Vertraulichkeit ihres Inhalts gewahrt bleibt. Die Benutzer können dann an datenschutzkonformen Analyseberechnungen teilnehmen, bei denen sichergestellt ist, dass ihr Logbuch aktuell und authentisch ist. Um eingeschränkten Benutzergeräten wie Smartphones gerecht zu werden, können Benutzer komplexere Analyseberechnungen an einen (möglicherweise böswilligen) Proxy auslagern, der nicht mit dem Systembetreiber zusammenarbeitet. Leistungsbewertungen unseres Prototyps zeigen, dass PUBA ausreichend Leistung bietet um Logbücher für die letzten 10-30 Transaktionen zu unterstützen.

In unserer zweiten Lösung namens *POBA* werden die Logbücher stattdessen auf von den Betreibern kontrollierten Servern gespeichert. Wir modellieren ein Setting, in dem mehrere Betreiber zusammenarbeiten, um das System zu betreiben, ohne sich gegenseitig vollständig zu vertrauen. Der Inhalt der Logbücher wird geschützt, indem er unter allen beteiligten Betreibern aufgeteilt wird. Darüber hinaus werden fortschrittliche kryptografische Tools wie Oblivious RAM eingesetzt, um die Identität der Benutzer zu schützen und zu verhindern, dass mehrere Interaktionen mit demselben Benutzer in Verbindung gebracht werden können. Da die Daten in dieser Umgebung ohne Benutzerinteraktion verfügbar sind, haben die Betreiber mehr Flexibilität bei der Durchführung von Analysen. Solange sich einige Betreiber ehrlich verhalten, stellt die Notwendigkeit aller Betreiber, den Berechnungen zuzustimmen, dennoch sicher, dass die Analyseergebnisse den Datenschutzanforderungen entsprechen. Leistungsbewertungen unseres Prototyps zeigen seine Praktikabilität im 3-Parteien Setting: Mit drei Betreibern können über zwei Millionen Logbucheinträge pro Tag verarbeitet werden.

Acknowledgements

First and foremost, I would like to thank Jörn Müller-Quade for giving me the opportunity to undertake this journey. Jörn provided me with a great environment to conduct my research in, with all the freedom I needed to carve out my path. I'm also very grateful to Markulf Kohlweiss for taking the time to serve as second referee of this thesis.

Second, I want to thank all my co-authors. It was great working with you and I hope for more great work together in the future.

Third, I want to thank all my current and former collegues that contributed to the very welcoming and supportive environment in our group: Dennis Hofheinz, Geoffroy Couteau, Alexander Koch, Gunnar Hartung, Jessica Koch, Lisa Kohl, Thomas Agrikola, Sven Maier, Akin Ünal, Bogdan Ursu, Julia Kastner, Rebecca Schwerdt, Jeremias Mechler, Michael Klooß, Valerie Fetzer, Lukas Beeck, Astrid Ottenhues, Felix Dörre, Marcel Tiepelt, Clemens Fruböse, Robin Berger, Laurin Benz, Christian Martin, Robert Brede, Eva Hetzel, Saskia Bayreuther, Yufan Jiang, Alexander Kühn, André Sperrle, Modjtaba Gharibyar and Julien Meier. Special thanks also to Willi Geißelmann, Carmen Manietta and Holger Hellmuth for their tireless effort. In particular, I'd like to thank Michael, Valerie and Marc Leinweber for many fruitful discussions, both about research and other topics.

Last but definitely not least, I want to express my deepest gratitute to my parents Wolfgang and Marlene, who already fostered my curiosity as a small child and were always supportive of me. Without you this journey would at the very least have been much more difficult.

Contents

1. Introduction

Cryptography has long been a cornerstone of secure communication, enabling parties to exchange messages confidentially even in the presence of adversaries. For most of its existence, cryptography dealt with securing the content of messages from outside adversaries. While initially firmly in the domain of states and military, message encryption has now become ubiquitous. Most internet traffic today is encrypted,[1] and modern messaging apps such as WhatsApp, Signal, or Telegram feature end-to-end encryption and guarantee the authenticity of received messages.

However, as digital systems have evolved, so too have the threats to user privacy. The focus of cryptographic research has gradually shifted from merely encrypting messages to protect against outside threats to designing protocols that operate securely even when some participants may be dishonest or compromised. This paradigm, often referred to as "cryptography in the presence of distrust," encompasses techniques such as secure multi-party computation and zero-knowledge proofs. These protocols enable collaborative data processing and analytics without requiring full trust in any single party, thereby reducing the risk of data misuse.

In our modern digital world, we constantly interact with digital systems and in doing so produce vast amounts of data, some of which can be quite sensitive. As an example, open loop tap-in/tap-out payment systems for public transportation, where users pay simply by tapping their credit card when entering and exiting vehicles, are convenient. Examples of such systems are transport for London[2] or transport for NSW.[3] These systems typically apply discounts for multi-vehicle trips, such as combining the tram with the ferry in NSW, and have daily/weekly/monthly caps corresponding to the prices of day tickets/weekly tickets/monthly tickets. However, each interaction, i.e., tapping the credit card, connects the user's identity with the time and place of that interaction, yielding extensive movement profiles that can be used to infer private attributes of the user [Kon+20].

Similarly, customer loyalty cards such as Payback [PAY20], Nectar [Aim20], or Optimum [Lob20] are widespread. And again, each use of such a loyalty card links the user's identity with the time and place, as well as the specific items bought. This information can again be used to infer sensitive information about the user, e.g., a pregnancy [Win20].

Collecting and storing large amounts of personal data is not only a problem for the users whose privacy is affected, but is also more and more becoming a problem for the

[1] https://transparencyreport.google.com/https/overview
[2] https://tfl.gov.uk/fares/how-to-pay-and-where-to-buy-tickets-and-oyster/pay-as-you-go
[3] https://transportnsw.info/tickets-opal/opal/contactless-payments

companies as well: Stronger privacy laws such as the *General Data Protection Regulation* (GDPR) [EPC16] in the EU put constraints on how and what personal information can be processed and threaten large fines for privacy violations. Since the introduction of the GDPR, other countries have enacted similar privacy laws, such as the *Lei Geral de Proteção de Dados Pessoais* (LGPD) [BR 18] in Brazil, the *Digital Personal Data Protection Act* (DPDP) [Ind23] in India, the *California Consumer Privacy Act* (CCPA) [CA 18] in California, USA, and many more. Thus, minimizing the data that is being collected while maintaining the desired utility is beneficial both for companies and users.

We want to investigate systems that consist of three main parts: A **data collection** step during which authenticated data is gathered, a **data storage** system to securely and privately store the collected data, and a **data analysis** system which allows privacy-preserving analysis computations on the collected data.

The main question we seek to answer in this thesis is:

> *Can we use cryptography to obtain generic building blocks that allow secure and private operation of a variety of such digital systems while offering practical performance?*

We will focus on the following scenario in this thesis: There is some digital system, operated by one or more system operators, that users interact with. Users register an account with the system. In the **data collection** step, user interactions with the system produce data that in itself is not sensitive, but linking it to the individual user makes it sensitive. This link between interactions and an identifiable user is needed for the functionality of the system. Since linking interactions (and thereby the collected data) to users results in sensitive data, users should be able to anonymously interact with the system in the data collection step. The collected data should then be available for **data analysis** by the system operator(s), and this analysis should be able to use the sensitive connection between user identity and data, as long as the result satisfies the desired privacy properties. This implies that all such analyses should be performed using some kind of secure computation. For the **data storage** system, we investigate both a solution where all data stays on user-controlled devices, as well as a solution where data is securely stored on operator-controlled servers.

This scenario applies to many deployed systems, such as customer loyalty systems, digital payment systems, check-in based mobility-as-a-service, and health insurance benefits programs.

Of course, if we want to talk about secure, private, and practical systems, this first needs suitable definitions of secure, private and practical. We call a system *secure* if it can provide the intended functionality even in the presence of malicious parties, including users or operators of the system, and when the building block is deployed in arbitrary context. For *privacy*, we require that the amount of sensitive information that can be learned about users is minimal (compared to what is strictly needed to achieve the desired functionality in an ideal world). Regarding *practical performance*, we fully implement our protocols and evaluate their performance, especially with regard to deployability within realistic

constraints, such as real-world user devices (smartphones) and acceptable end-to-end latencies for user interactions (e.g., participating in a customer loyalty program should not take longer than executing a typical payment transaction).

1.1. Contribution of the Thesis

When collecting user data, there are two architectural options regarding data storage: either on the *user's side*, i.e., each user keeps track of their own collected data, or on the *operator's side*. We now take a closer look at both options.

If data is stored on the user's side until analysis, the user has full control over their own data at all times. Users must explicitly cooperate and provide their data to the secure multi-party computation protocol each time data analyses are to be performed. This is great for use-cases where analysis is performed rarely or on very sensitive data, however, for frequently recurring tasks such as usage-based invoice generation or general usage statistics, this can be impractical. Furthermore, users are responsible for the availability of their data, i.e., a lost smartphone on which the collected data was stored might result in the need to pay the maximum amount in a usage-based fee model. Additionally, mechanisms to ensure the authenticity and "freshness" of the collected data during analysis are required to prevent users from selectively omitting some data points when providing their collected data. This results in the need for heavier cryptographic components on the user's side, which impact performance for users who may only have a resource-constrained device.

When data is stored on the operator's side, users must give up control over their data, but are no longer responsible for securing their data. However, it is now imperative to ensure that the data is stored securely and cannot be extracted in plaintext by an operator. This can be achieved for example through secret-sharing the data between the operators and performing the analysis with MPC, or by using fully homomorphic encryption (FHE). Since users do not have direct control over how their data is used in this approach, it is important that the protocol allows only functions approved by the consortium of operators to be evaluated on the data, and that this consortium has enough members that will object to the evaluation of analyses that violate user privacy (see Section 4.2.1 for further discussion).

In this thesis, we construct and evaluate protocols in both settings.

1.1.1. User-Side Data Storage

The first part of this thesis in Chapter 3 covers the approach of storing collected data locally on each user's device. This data can then be submitted to analytics computations, and the mechanisms used during collection ensure authenticity of the data.

To this end, we designed PUBA, a universally composable building block with local data storage: A logbook (solely) kept on the user's device should be updated with the observed data such that its content cannot be manipulated by the user while its current

content cannot be learned by the operator. The former is particularly important for requirements like fraud or money laundering detection. More precisely, we need to ensure the authenticity, integrity, freshness, confidentiality, and unlinkability of the logbook. At the same time, the building block needs to allow for analytical computations, where users are incentivized or forced to take part in privacy-preserving analytics of their latest logbook. These computations must not leak the input(s), but the operator may only obtain some privacy-preserving statistics, classification, etc. as output. The user might also receive some output, e.g., targeted ads in case of a loyalty system. Also, in some cases, a slight update of the logbook with results from the analytics computation might be required, e.g., a risk level resulting from the background fraud detection checks. Depending on the scenario, the actual analytics function might need to be kept confidential, e.g., when it is considered intellectual property like the classifier for targeted advertising. In this case, it needs to be ensured that the operator cannot misuse this opportunity to selectively deploy privacy-violating functions.

We distinguish two types of analytics computations: outsourced heavyweight analytics and direct lightweight analytics. As the privacy-preserving evaluation of machine-learning classifiers (e.g., to determine a user's individual fraud risk level) might be computationally and communicationally very demanding, in particular when mobile user devices are involved, there needs to be a possibility to securely outsource this computation to a proxy server who acts on behalf of the user(s). Ideally, even a malicious proxy should not learn anything about user and operator inputs, as well as the output provided to the user(s). In some scenarios, outsourced analytics should not block further interactions with the system, i.e., users should be able to continue updating their logbooks while analytics is running. For instance, in loyalty systems customers should be able to continue shopping and keep track of purchased items while the outsourced ad selection process (for a previous version of the logbook) is still in progress. Direct lightweight privacy-preserving analytics is jointly performed between the user's device and the operator. This type of analytics might be done prior to an update of the logbook to determine further actions, e.g., whether the payment transaction just initiated should be accepted in view of the customer's personal risk level.

PUBA's user-facing protocols are fully implemented and practical evaluation on a smartphone as user device and simple server hardware for operator and proxy demonstrate practicability for modest-sized logbooks.

1.1.2. Operator-Side Data Storage

The second part of this thesis in Chapter 4 covers the approach of storing the collected data centrally on operator-controlled servers. In this setting, multiple operators that jointly operate the system are required.

We designed POBA as a universally composable building block. Then, the sensitive user data can be securely distributed between these operators through the means of a secret-sharing based database. Since the sensitive user data can be used without any

user interaction, it is important to have multiple operators where each user trusts that at least a certain threshold of them behaves honestly. Entries for user logbooks can only be generated through agreement of the involved (anonymous) user and operator on the content, and insertion is performed in a way that ensures unlinkability both towards the user as well as with other entries of that user. This process is also designed in a way to work offline (only a connection between the user device and some operator-controlled terminal is required, but no connection to all operators or centrally managed state is required).

To ensure that the outputs of evaluated analysis functions preserve the desired level of privacy, we require consensus among the operators on what functions to evaluate. Assuming a suitable threshold of operators behaves honestly, this ensures that only acceptable functions are being evaluated on sensitive data.

Again, POBA's protocols are fully implemented. Practical evaluation shows that in the special case of three operators, the system can scale to millions of users, while in the more general case of an arbitrary number of operators it is limited to some tens of thousands of users.

1.2. Other Results

This section briefly discusses other work published during my doctoral studies that is not included in this thesis.

1.2.1. Black-Box Accumulators: A Simple Form of Privacy-Preserving Data Collection ([Hof+20b; Fal+21b])

Black-box accumulators (BBA), introduced by Jager and Rupp [JR16] and extended by Hartung, Hoffmann, Nagel, and Rupp [Har+17] allow privacy-preserving point collection and redemption. In [Hof+20b] building on the ideas of [Har+17], we designed a very efficient scheme for prepayments, called black-box wallets. We improve privacy by hiding the user balance during redemption, which is essential for prepaid payments. [Har+17] concluded that hiding the balance was too expensive and instead reveal the balance in every redeem interaction. Our system employs a novel blind signature construction that allows efficient zero-knowledge proofs without the need of pairings, which together with state-of-the-art rangeproofs results in an extremely efficient protocol. This enables use on a wide range of low-power smartphones and smartwatches and can plausibly be used on smartcards.

In [Fal+21b], we propose a black-box accumulator scheme based on lattice assumptions, which is the first plausibly post-quantum secure implementation of BBA.

1.2.2. Auditable Trapdoors ([Fet+23b; Ste+21b])

In privacy-preserving protocols, it is often necessary to de-anonymize users under certain conditions, for example due to legal constraints or after detecting misuse. In [Fet+23b], we designed a system to conditionally revoke anonymity of users while safeguarding against silent misuse of this feature. Concretely, we show how to use anonymous committees to manage a decryption key and enforce policies for the usage of this decryption key, using a public bulletin board to leave auditable traces of each decryption request. Then, we show how to create user-specific short-term trapdoors that are encrypted under this key and which can be used to de-anonymize specific users during specific timeframes. Additional auditing information is made available to a designated auditor party, and general statistics regarding frequency and amount of deanonymization requests are publicly available.

In an earlier work [Ste+21b], we showed how to design a simpler system that distributes a decryption key among a blockchain-based committee which can later on-demand grant access to the content of ciphertexts. We designed a smart contract based on the Ethereum blockchain that handles key generation and reconstruction and evaluated its practicability.

1.2.3. Composable Longterm Privacy ([Ber+23b])

When handling sensitive data, especially personal data such as medical information, an important question is how privacy can be guaranteed in the long term, even if new developments such as quantum computers or other cryptanalytic advances break currently used cryptographic building blocks. In [Ber+23b], we investigate how to enable composable longterm privacy, mainly in the special case of commitments. This is applicable to the protocol in Chapter 3, where commitments play a central role, although the proposed commitment is more of theoretical relevance and has poor practical performance.

1.3. Structure of the Thesis

This thesis is structured in three main parts: The first part, consisting of this chapter and Chapter 2, introduces the problem of user data collection and analysis as well as the needed cryptographic building blocks.

The second part contains the main contribution of the thesis. In Chapter 3 we present our new composable framework *PUBA* for data collection on user devices that can be employed by a single provider with the help of some untrusted but non-colluding third party. In Chapter 4 we then present our new composable framework *POBA* for data collection on operator devices that supports three or more providers without the need of an additional external party. In Chapter 5, we compare the two presented solutions and detail some open questions.

Finally, the last part consists of the appendix, which mainly includes the detailed security proofs for the main contributions. Chapter A corresponds to Chapter 3 and Chapter B corresponds to Chapter 4.

2. Preliminaries

In this chapter, we introduce some general notation and give formal definitions of all the cryptographic building blocks that are used in the following chapters. Additionally, we give a short introduction to the Universal Composability framework, in which we conduct all our security analyses.

The definitions in this chapter are taken mostly verbatim from [Fau+25c; Fet+22b] unless otherwise noted.

2.1. Notation

We use $\coloneqq$ to denote deterministic assignment and $\leftarrow$ for probabilistic sampling. If $\mathcal{X}$ is a set, $x \leftarrow \mathcal{X}$ denotes that x is sampled uniformly at random from $\mathcal{X}$. If alg is a probabilistic algorithm, $x \leftarrow \mathsf{alg}(y)$ denotes running alg on input y with freshly sampled randomness and assigning the output to x. If we want to make the randomness explicit, we will instead write $x \leftarrow \mathsf{alg}(y; r)$, which implies that r is fresh randomness. We denote the probability of an event X occurring with $\Pr[X]$. We use the standard notation for mathematical sets such as $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Z}_p$ or $\mathbb{F}_p$.

We denote the security parameter with λ, and typically assume it is given to all algorithms in unary notation, denoted by 1^λ. A function $\mathsf{negl}(\lambda)$ is called negligible if for every polynomial p, there exists $n_0 \in \mathbb{N}$ so that for all $n > n_0$ $negl(n) < \frac{1}{p(n)}$. We call an algorithm probabilistic polynomial time (PPT) if it can consume randomness in addition to its input and there is some fixed polynomial in the security parameter that bounds its running time.

With $\{0,1\}^n$ we denote the set of all binary strings of length n, and with $\{0,1\}^*$ the set of all finite-length binary strings.

2.2. The Universal Composability Framework

We performed our analysis in the Universal Composability (UC) framework [Can01; Can20]—a common framework for investigating the security of cryptographic protocols. It follows the simulation-based paradigm: A protocol Π in the *real world* is shown to be *as secure as* the same computation performed by a trusted party who, in a (hypothetical) *ideal world*, follows the description provided by an ideal functionality $\mathcal{F}$. This is generally

modeled by a simulator's ability to *simulate* the real world given only information accessible in the ideal world such that no environment $\mathcal{Z}$ can distinguish the two worlds. The environment picks the inputs for all parties and decides which of them to corrupt—in which case they are controlled by the environment. The simulator on the other hand is unaware of the parties inputs and only obtains information provided by $\mathcal{F}$.

UC provides strong composition guarantees; a protocol proven secure remains secure regardless which other protocols are executed in between two consecutive messages. This requires the simulator to deliver messages *in time* and waiting for the environments response instead of only having to provide a transcript for the entire protocol execution as in the stand-alone setting [GM82; GMW87].

The key idea of simulation based security can be summarized as follows: Secure multi-party computation (MPC) is relatively easy if we have a single entity that is trusted by all the participants and with whom the parties can communicate over secure and authenticated channels. The behavior of this party is referred to as the ideal functionality which defines how the inputs are used for computations, which outputs exist and which party obtains which output. The ideal functionality can be only accessed in a black-box way: each party sends its inputs to the trusted entity and obtains its output, without learning any other information in the process. This also makes the level of privacy apparent as anything that an adversary can learn is modelled explicitly by having the trusted party send this information to the adversary. The ideal functionality is generally much easier to understand than the protocol and thus enables easier analysis of the security guarantees.

As it is unrealistic to assume that such a trusted party exists in the real world we refer to the world in which the computations are performed by the trusted party as the *ideal world*.

The actual protocol is then executed in the *real world* where (mutually distrustful) parties interact with each other according to the protocol description. If the real world can be shown to be *indistinguishable* to an interaction of all parties with the ideal functionality then the protocol inherits all the security guarantees of the ideal functionality. This requires a *simulator* that reports all protocol messages from the honest parties, as honest parties in the ideal world only forward their input to the functionality and do not send protocol messages. The simulator reports the messages without knowing the actual input and only learns the leaks provided by the ideal functionality. By proving that the view provided by the simulator in the ideal world is *indistinguishable* from the view of a protocol execution in the real world we get the guarantee that any attack on the protocol can also be launched against the ideal functionality. In particular, this means that no attacks on the real world exist that leak information which cannot be extracted in the ideal world.

While the standalone-setting requires the simulator to provide the entire transcript as a whole, the Universal Composability framework extends this scenario by requiring the simulator to provide the protocol messages *in time* and await the environments response. This means that while in the standalone-setting, the transcript can be arbitrarily rewritten during simulation as long as the result still looks like a valid protocol execution, in the UC-setting a message once reported can not be changed later. This models the idea that

Functionality $\mathcal{F}_{\text{CRS}}$

Parameters: A distribution $\mathcal{D}$

- On input (`value`) from party P:
  - If $d = \perp$, set $d \leftarrow \mathcal{D}$
  - Generate public delayed output (d) to party P

(a) Functionality $\mathcal{F}_{\text{CRS}}$ for common reference strings

Functionality $\mathcal{F}_{\text{BB}}$

- Upon receiving (`Register`, v) from some party P_i with pid pid_{P_i}:
  1. Check that no value (pid_{P_i}, v') is already recorded, otherwise ignore this input
  2. Record (pid_{P_i}, v)
- Upon receiving (`Retrieve`, pid) from some party P_j or the adversary $\mathcal{S}$:
  1. Check if some pair (pid, v) is recorded
  2. If yes:
     - Return (pid, v) to P_j or $\mathcal{S}$
  3. Otherwise:
     - Return $(pid, \perp)$ to P_j or $\mathcal{S}$

(b) Functionality $\mathcal{F}_{\text{BB}}$ for a public bulletin board

Figure 2.1.: Setup functionalities

other protocols are running in parallel and hence ensures that security guarantees remain valid in arbitrary environments instead of requiring the protocol to be blocking as in the standalone case.

Each session in the UC-framework has a unique session identifier (sid). We additionally assume that each new task is assigned with a unique (inside this session) subsession identifier (ssid).

UC provides no restrictions on the scheduling of protocols which results in a rather strong security guarantee. However, this comes at the cost of feasibility. In fact, it was shown in [CF01] that not even weak (MPC-incomplete) building blocks such as commitments can be proven secure in this framework without set-up assumptions. This is why most constructions in the UC framework rely on set-up assumptions such as a common reference string (CRS) which is assumed to be a hybrid functionality that is set up by a trusted party.

In this thesis, we make use of $\mathcal{F}_{\text{CRS}}$ as shown in Fig. 2.1a and a public bulletin board functionality $\mathcal{F}_{\text{BB}}$ as shown in Fig. 2.1b.

2.3. Secure Communication in UC

Communication in UC is usually built on top of authenticated or secure channels, which uniquely identify the participating parties and ensure authenticity or authenticity and confidentiality of messages sent, respectively. For communication between different system operators, or operators and other non-user participants, we use the standard functionality for secure channels, $\mathcal{F}_{\text{SMT}}$ as shown in Fig. 2.2a.

However, with the goal of allowing anonymous interactions between users and digital systems, we additionally require a different form of communication channel: One party, party A (which will typically be a user), should be able to initiate a secure channel with party B in a way where the identity of party B (typically a system operator or similar party) is verified, so party A has the guarantee that messages can only be read by party B,

Functionality $\mathcal{F}_{\text{SMT}}$

- On input $(ssid,\ pid_R,\ msg)$ from party S:
  - Leak $(ssid,\ pid_S,\ pid_R,\ |msg|)$ to the adversary
  - Generate private delayed output $(ssid, pid_S, msg)$ to party R with pid pid_R

(a) Functionality $\mathcal{F}_{\text{SMT}}$ for authenticated communication

Functionality $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$

State: List L of open channels

- On input $(\texttt{send},\ ssid,\ pid_R,\ msg)$ from party S:
  - Store $(ssid,\ pid_S,\ pid_R)$ in L
  - Leak $(ssid,\ \texttt{send},\ pid_R,\ |msg|)$ to the adversary
  - Generate private delayed output $(\texttt{send},\ ssid,\ msg)$ to party R with pid pid_R
- On input $(\texttt{respond},\ ssid,\ msg)$ from party R with pid pid_R:
  - Find entry $(ssid,\ pid_S,\ pid_R)$ in L with matching $ssid$ and pid_R, otherwise ignore this input
  - Leak $(ssid,\ \texttt{respond},\ pid_S,\ |msg|)$ to the adversary
  - Generate private delayed output $(\texttt{respond},\ ssid,\ msg)$ to party S with pid pid_S

(b) Functionality $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ for one-sided authenticated communication

Figure 2.2.: Functionalities for secure communication

and any message received over the channel comes from party B. Party B on the other hand only learns that some undisclosed party has sent a message (although it is guaranteed that the message was not modified) and can send messages back on the channel, with the guarantee that only the party that initiated the channel can read them, but without learning the identity of party A.

Such a channel is similar to how TLS works: A user connects to the server, verifies the certificate it receives to ensure the identity of the server matches, and then they can securely communicate. If stronger privacy is required (since normal TLS connections reveal the IP address of the user), either tools such as onion routing can be employed, or a different technology without revealing metadata such as near-field communication (NFC) or Bluetooth Low Energy (BLE) can be used.

Formally, we model our secure channel with one-sided authentification as the functionality $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ shown in Fig. 2.2b. Any party S can initiate a secure connection to some other party R by sending $(\texttt{send},\ ssid,\ pid_R,\ msg)$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$, which guarantees that the party with id pid_R will be the one receiving this message. Party R only receives the message, but no information about the sender. However, it can respond to the message by sending $(\texttt{respond},\ ssid,\ msg')$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$, using the same subsession id $ssid$ under which it received the message. $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ will then privately output msg' to party S together with the information that it came from party R.

2.4. Pairing Groups

We instantiate the constructions in this work over pairing groups. In general, we write $\mathbb{G}, \mathbb{H}$, etc., for groups and use capital letters G, H, etc., for group elements. All groups used are cyclic and we use *additive* notation, i.e., we write $G + H$ and $x \cdot G$ or xG for $G, H \in \mathbb{G}, x \in \mathbb{Z}$.

We call a tuple $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ a pairing group if there exists an efficient map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with the properties that:

- $e(G_1, G_2) = G_\mathrm{T}$ for generators $G_1 \in \mathbb{G}_1, G_2 \in \mathbb{G}_2, G_\mathrm{T} \in \mathbb{G}_T$
- $e(xG_1, yG_2) = xy \cdot e(G_1, G_2)$ for $G_1 \in \mathbb{G}_1, G_2 \in \mathbb{G}_2, x, y \in \mathbb{Z}$

We work over "type 3" pairing groups, where there are no efficient homomorphisms between $\mathbb{G}_1$ and $\mathbb{G}_2$ We sometimes write $G_1 \cdot G_2$ as a shorthand for $e(G_1, G_2)$.

A PPT algorithm GroupGen on input 1^λ outputs a (description of) a group ($\mathbb{G} = \mathbb{G}_\lambda$, G, $p = p_\lambda$) of prime order p with generator G. Given the description, group operations (addition, inverse) and membership tests are efficient.

A PPT algorithm PairingGen on input 1^λ outputs a (description of) groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order p and generators G_1, G_2, G_T together with a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Given the description, group operations (addition, inverse) and membership tests are efficient in all three groups and the bilinear map e can be computed efficiently.

Definition 2.4.1 (Discrete Logarithm Problem (DLOG))**.** The DLOG assumption in a group $\mathbb{G}$, or more specifically for a group generator GroupGen, holds if for every PPT adversary $\mathcal{A}$ the advantage

$$\mathsf{Adv}^{\mathsf{DLOG}}_{\mathcal{A}}(1^\lambda) = \Pr\left[\begin{array}{l} \mathbb{G} \leftarrow \mathsf{GroupGen}(1^\lambda); G \leftarrow \mathbb{G} \setminus \{0\}; H \leftarrow \mathbb{G}; \\ x \leftarrow \mathcal{A}(1^\lambda, \mathbb{G}, G, H) : xG = H \end{array}\right]$$

is negligible.

For simplicity, we leave GroupGen (and PairingGen) implicit in the following.

Definition 2.4.2 (Computational Diffie-Hellman Problem (CDH))**.** The CDH assumption in a group $\mathbb{G}$ holds if for every PPT adversary $\mathcal{A}$ the advantage

$$\mathsf{Adv}^{\mathsf{CDH}}_{\mathcal{A}}(1^\lambda) = \Pr\left[\begin{array}{l} G \leftarrow \mathbb{G} \setminus \{0\}; x, y \leftarrow \mathbb{Z}_p; \\ Z \leftarrow \mathcal{A}(1^\lambda, \mathbb{G}, G, xG, yG) : Z = xyG \end{array}\right]$$

is negligible.

Definition 2.4.3 (Decisional Diffie-Hellman Problem (DDH))**.** The DDH assumption in a group $\mathbb{G}$ holds if for every PPT adversary $\mathcal{A}$ the advantage

$$\mathsf{Adv}^{\mathsf{DDH}}_{\mathcal{A}}(1^\lambda) = \left|2 \cdot \Pr\left[\begin{array}{l} G \leftarrow \mathbb{G} \setminus \{0\}; x, y \leftarrow \mathbb{Z}_p; Z_0 = xyG; Z_1 \leftarrow \mathbb{G}; \\ b \leftarrow \{0,1\}; b' \leftarrow \mathcal{A}(1^\lambda, \mathbb{G}, G, xG, yG, Z_b) : b = b' \end{array}\right] - 1\right|$$

is negligible.

Definition 2.4.4 (Symmetric External Diffie-Hellman Problem (SXDH))**.** The SXDH assumption in a pairing group $\mathsf{gp} := (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, G_1, G_2, G_t, e)$ holds if the DDH assumption holds in both $\mathbb{G}_1$ and $\mathbb{G}_2$.

Definition 2.4.5 (Co-CDH problem)**.** The Co-CDH assumption in a pairing group $\mathsf{gp} \coloneqq (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, G_1, G_2, G_t, e)$ holds if for every PPT adversary $\mathcal{A}$ the advantage

$$\mathsf{Adv}^{\mathsf{CO\text{-}CDH}}_{\mathcal{A}}(1^\lambda) = \Pr\left[\ x \leftarrow \mathbb{Z}_p; X \leftarrow \mathcal{A}(1^\lambda, \mathsf{gp}, xG_1) : X = xG_2\ \right]$$

is negligible.

The Co-CDH assumption is implied by the SXDH assumption.

2.5. Zero-Knowledge Proofs

Zero-knowledge proofs allow a prover to convince a verifier of the truthfulness of some statement without revealing any information besides the fact that the statement is true. In this work, we are mainly interested in so-called proofs of knowledge (PoK), also called extractable zero-knowledge proofs. A PoK is defined relative to some NP relation R, and a prover not only convinces a verifier that a statement is true (i.e., the statement is in the language defined by R), but also that it *knows* a witness for this. Knowledge is formalized as the existence of an extractor that, given black-box access to such a prover, can extract a witness with probability similar to the probability with which the prover convinces a verifier.

Non-interactive zero-knowledge proofs of knowledge (NIZKPoK) cannot exist in the plain model. In this thesis, we only define NIZKPoKs in the common reference string (CRS) model. We will often use NIZK or just ZK as a shorthand for a NIZPoK.

Definition 2.5.1 (Non-Interactive Proof System)**.** A *non-interactive proof system* ZK for NP-relation R is a tuple $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Vf})$ of PPT algorithms, where

- $\mathsf{Setup}(1^\lambda)$ outputs a common reference string *crs*
- $\mathsf{Prove}(crs, stmt, wit)$ generates a proof π given $(stmt, wit) \in R$.
- $\mathsf{Vf}(crs, stmt, \pi)$ verifies a proof π for statement *stmt* and outputs 0 or 1.

It is a *NIZKPoK* if additionally there are algorithms $\mathsf{ZK.Sim}$ and $\mathsf{ZK.Ext}$ with

Note that we sometimes explicitly add the relation R as input to the algorithms to make it clear for which relation the NIZPoK is.

Zero-knowledge is defined by a simulator that, given a trapdoor generated together with the common reference string (CRS), can generate proofs without knowing a witness. For this, we first require that an adversary cannot distinguish the real CRS from a simulated one. Then, we also require that an adversary cannot distinguish real proofs from simulated proofs. For composable zero-knowledge, we require that real and simulated proofs are indistinguishable, *even when giving the simulation trapdoor to the adversary.*

Definition 2.5.2 (Composable Zero-Knowledge [GS08; Gro06])**.** A non-interactive proof system ZK for relation R is called *zero-knowledge* if there exist PPT algorithms $(\mathsf{ZK.SimSetup}, \mathsf{ZK.Sim})$, where

- $\mathsf{SimSetup}(1^\lambda)$ outputs $(crs,\ td_{\mathsf{sim}})$, where td_{sim} is a simulation trapdoor
- $\mathsf{Sim}(crs,\ stmt,\ td_{\mathsf{sim}})$ generates a proof π for the statement $stmt$ given the simulation trapdoor td_{sim} for the crs.

We require indistinguishability of the common reference strings:

$$\Pr\left[\ crs \leftarrow \mathsf{Setup}(1^\lambda); \mathcal{A}(crs) = 1\ \right] - \Pr\left[\ (crs,\ td_{\mathsf{sim}}) \leftarrow \mathsf{SimSetup}(1^\lambda); \mathcal{A}(crs) = 1\ \right] \leq \mathsf{negl}(\lambda)$$

Additionally, we require the following distributions to be the same:

$$\mathrm{Real}_{\mathcal{A}}(1^\lambda) = \Pr\left[\begin{array}{l}(crs,\ td_{\mathsf{sim}}) \leftarrow \mathsf{SimSetup}(1^\lambda); (stmt,\ wit) \leftarrow \mathcal{A}(crs,\ td_{\mathsf{sim}});\\ \pi \leftarrow \mathsf{Prove}(crs,\ stmt,\ wit); b \leftarrow \mathcal{A}(\pi) : b \wedge R(stmt,\ wit) = 1\end{array}\right]$$

$$\mathrm{Sim}_{\mathcal{A}}(1^\lambda) = \Pr\left[\begin{array}{l}(crs,\ td_{\mathsf{sim}}) \leftarrow \mathsf{SimSetup}(1^\lambda); (stmt,\ wit)\mathcal{A}(crs,\ td_{\mathsf{sim}});\\ \pi \leftarrow \mathsf{Sim}(crs,\ stmt,\ td_{\mathsf{sim}}); b \leftarrow \mathcal{A}(\pi) : b \wedge R(stmt,\ wit) = 1\end{array}\right]$$

More precisely, for all PPT adversaries $\mathcal{A}$, we require $\mathrm{Real}_{\mathcal{A}}(1^\lambda) = \mathrm{Sim}_{\mathcal{A}}(1^\lambda)$.

As mentioned before, proof of knowledge is defined by giving an extractor that, again given a trapdoor generated together with the common reference string, can extract a witness given a statement and accepting proof. Again, crs with trapdoor and honest crs should be indistinguishable.

Definition 2.5.3 (Proof of Knowledge)**.** A non-interactive proof system ZK for relation R is called a *proof of knowledge* if there exist PPT algorithms $(\mathsf{ZK.ExtSetup},\ \mathsf{ZK.Ext})$, where

- $\mathsf{ExtSetup}(1^\lambda)$ outputs $(crs,\ td_{\mathsf{ext}})$, where td_{ext} is an extraction trapdoor
- $\mathsf{Ext}(crs,\ stmt,\ \pi,\ td_{\mathsf{ext}})$ outputs a witness wit such that if $\mathsf{Vf}(crs,\ stmt,\ \pi) = true$ it holds that $R(stmt,\ wit) = 1$, except with negligible probability.

Again, we require the following distributions to be indistinguishable:

$$\mathrm{Real}_{\mathcal{A}}(1^\lambda) = \Pr\left[\begin{array}{l}crs \leftarrow \mathsf{Setup}(1^\lambda); (b,\ stmt,\ \pi) \leftarrow \mathcal{A}(crs) :\\ b \wedge \mathsf{Vf}(crs,\ stmt,\ \pi) = true\end{array}\right]$$

$$\mathrm{Ext}_{\mathcal{A}}(1^\lambda) = \Pr\left[\begin{array}{l}(crs,\ td_{\mathsf{ext}}) \leftarrow \mathsf{ExtSetup}(1^\lambda); (b,\ stmt,\ \pi) \leftarrow \mathcal{A}(crs);\\ wit \leftarrow \mathsf{Ext}(crs,\ stmt,\ \pi,\ td_{\mathsf{ext}}) :\\ b \wedge \mathsf{Vf}(crs,\ stmt,\ \pi) = true \wedge R(stmt,\ wit) = 1\end{array}\right]$$

More precisely, we define the advantage of $\mathcal{A}$ as $\mathsf{Adv}^{\mathsf{Ext}}_{\mathcal{A}} \coloneqq \mathrm{Real}_{\mathcal{A}}(1^\lambda) - \mathrm{Ext}_{\mathcal{A}}(1^\lambda)$. Then, ZK is called a proof of knowledge, or alternatively straight-line extractable, if for all PPT $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\lambda)$ such that $\mathsf{Adv}^{\mathsf{Ext}}_{\mathcal{A}} \leq \mathsf{negl}(\lambda)$. It has statistical extractability if the same holds for all $\mathcal{A}$ (not only PPT), and perfect extractability if $\mathsf{Adv}^{\mathsf{Ext}}_{\mathcal{A}} = 0$ for all $\mathcal{A}$.

Soundness of a NIZK (i.e., the inability for an adversary to produce proofs for false statements) follows from extractability, with the soundness error bounded by the advantage $\mathsf{Adv}^{\mathsf{Ext}}_{\mathcal{A}}$.

A NIZK proof system that is either zero-knowledge or extractable, depending on how the common reference string is created, is called a dual-mode NIZK.

With a dual-mode NIZK as described above, it is only possible to either simulate proofs or extract witnesses for a given crs, but not both. This is fine in the setting of Chapter 3. However, in settings where we need to both simulate proofs of honest parties and extract witnesses from corrupted parties using the same crs, as in Chapter 4, we need a stronger property: simulation-extractability. For this, a simulated crs must be set up with both a simulation trapdoor and an extraction trapdoor together. Not only does this enable simulation and extraction for the same crs, it also ensures that witness extraction succeeds even if the adversary has access to simulated proofs.

Definition 2.5.4 (Straight-Line Simulation-Extractability)**.** Let ZK be a NIZKPoK for NP-relation R, and let there be additional PPT algorithms $(\mathsf{ZK.SimSetup},\ \mathsf{ZK.Sim},\ \mathsf{ZK.Ext})$, where

- $\mathsf{SimSetup}(1^\lambda)$ outputs $(crs,\ td_{\mathsf{sim}},\ td_{\mathsf{ext}})$, where td_{sim} is a simulation trapdoor and td_{ext} is an extraction trapdoor.
- $\mathsf{Sim}(crs,\ stmt,\ td_{\mathsf{sim}})$ generates a proof π for the statement $stmt$ given the simulation trapdoor td_{sim} for the crs.
- $\mathsf{Ext}(crs,\ stmt,\ \pi,\ td_{\mathsf{ext}})$ outputs a witness wit such that if $\mathsf{Vf}(crs,\ stmt,\ \pi) = true$ it holds that $R(stmt,\ wit) = 1$, except with negligible probability.

We say ZK is *straight-line simulation-extractable* if the following holds:

1. Indistinguishability of crs:

$$\begin{aligned} &\Pr\left[\ crs \leftarrow \mathsf{Setup}(1^\lambda); \mathcal{A}(crs) = 1\ \right] \\ -&\Pr\left[\ (crs,\ td) \leftarrow \mathsf{SimSetup}(1^\lambda); \mathcal{A}(crs) = 1\ \right] \leq \mathsf{negl}(\lambda) \end{aligned}$$

2. Composable Zero-Knowledge:
   ZK together with ZK.Sim and ZK.SimSetup′ is composable zero-knowledge according to Definition 2.5.2, where $\mathsf{ZK.SimSetup}'(1^\lambda)$ runs $(crs,\ td_{\mathsf{sim}},\ td_{\mathsf{ext}}) \leftarrow \mathsf{ZK.SimSetup}(1^\lambda)$ and outputs $(crs,\ td_{\mathsf{sim}})$
3. For any PPT adversary $\mathcal{A}$, the probability that it wins the following game is negligible:
   a) Setup $(crs,\ td_{\mathsf{sim}},\ td_{\mathsf{ext}}) \leftarrow \mathsf{SimSetup}(1^\lambda)$.
   b) Run $\mathcal{A}^{O_{\mathsf{sim}}}(1^\lambda,\ crs)$, where the oracle O_{sim} is defined as:
      - $O_{\mathsf{sim}}(stmt)$: $\pi \leftarrow \mathsf{Sim}(crs,\ stmt,\ td_{\mathsf{sim}})$, store $(stmt,\ \pi)$ in Q, return π
   c) Eventually, $\mathcal{A}$ outputs $(stmt^*,\ \pi^*)$

d) $\mathcal{A}$ wins the game if:

- $(stmt^*,\ \pi^*) \notin Q$
- $\mathsf{Vf}(crs,\ stmt^*,\ \pi^*) = true$
- $wit \leftarrow \mathsf{Ext}(crs,\ stmt^*,\ \pi^*,\ td_{\mathsf{ext}})$: $R(stmt^*,\ wit) = 0$

The Groth–Sahai Proof System and Structure-Preserving Primitives We build the protocols in this thesis around the non-interactive zero-knowledge proof system of Groth and Sahai [GS08] and the extension of Escala and Groth [EG14]. The Groth–Sahai proof system is built around a pairing group $(\mathbb{G}_1,\ \mathbb{G}_2,\ \mathbb{G}_T)$ and can efficiently prove statements consisting of pairing-product equations. These are equations of the form

$$\mathbf{X}\Gamma\mathbf{Y} = t_T$$

where $\mathbf{X} = (X_1,\ \ldots,\ X_m) \in \mathbb{G}_1^m$, $\mathbf{Y} = (Y_1,\ \ldots,\ Y_n) \in \mathbb{G}_2^n$ are vectors of group elements, where each element can be either a public constant or a secret witness value, and $\Gamma = \{\gamma_{ij}\}_{i=1,j=1}^{m,n} \in \mathbb{Z}_p^{m\times n}$ and $t_T \in \mathbb{G}_T$ are public values. Thus, our goal is to pick our other building blocks in a way to be compatible with Groth–Sahai proofs, i.e., such that all equations we want to prove are pairing-product equations.

Building blocks that satisfy this property are called structure-preserving, introduced by [Abe+16]. On a high level, a building block is structure-preserving if all public objects (such as common reference strings, public keys, messages, signatures, commitments, ciphertexts, etc.) only consist of group elements in $\mathbb{G}_1$ and $\mathbb{G}_2$ and all algorithms of interest (such as signature verification, commitment opening, etc.) only need to check membership in $\mathbb{G}_1$ and $\mathbb{G}_2$ and evaluate relations described by pairing product equations.

We will provide formal definitions of what it means to be structure-preserving for each relevant primitive individually in their respective section.

2.6. Secret-Sharing

A (n, t) secret-sharing scheme consists of a sharing algorithm that, given some secret, outputs n shares such that up to t shares reveal no information about the secret, and a reconstruction algorithm that given at least $t + 1$ shares reconstructs the secret.

Definition 2.6.1 (Syntax of a Secret-Sharing Scheme)**.** A (n, t) secret-sharing scheme S with number of parties n and corruption threshold t is a tuple $\mathsf{S} = (\mathsf{S.Share}, \mathsf{S.Recon})$ of PPT algorithms, where

- $\mathsf{S.Share}(m)$ outputs shares $\langle m\rangle_1, \ldots, \langle m\rangle_n$ such that up to t shares are information-theoretically independent of m

- $\mathsf{S.Recon}(\langle m \rangle_1, \ldots, \langle m \rangle_n)$ outputs m if each $\langle m \rangle_i$ is either $\perp$ or the output of $\mathsf{S.Share}(m)$, and at least $t + 1$ shares are not $\perp$

We will refer to $\langle m \rangle$ as a S-sharing of message m.

2.7. Encryption

Encryption of messages has been the main focus of cryptography for most of its history.

Encryption schemes come in two main flavours: A symmetric key version, where both the encrypting and decrypting party share the same secret key, and an asymmetric key version, also called public key encryption (PKE). We do not make explicit use of symmetric encryption in this thesis, so only definitions for asymmetric encryption are given.

2.7.1. Public Key Encryption (PKE)

A public key encryption scheme features two keys, the public encryption key which allows encryption, and the secret decryption key which allows decryption.

Definition 2.7.1 (Syntax of PKE)**.** A PKE scheme PKE with message space $\mathcal{M}$ consists of a tuple $(\mathsf{Keygen}, \mathsf{Encrypt}, \mathsf{Decrypt})$ of PPT algorithms such that

- $\mathsf{Keygen}(1^\lambda)$ outputs a public encryption key ek and a secret decryption key dk
- $\mathsf{Encrypt}(\mathsf{ek}, m)$ takes an encryption key ek, a message m in the message space $\mathcal{M}$, and outputs a ciphertext ct
- $\mathsf{Decrypt}(\mathsf{dk}, ct)$ takes a decryption key dk, a ciphertext ct and outputs a message $m \in \mathcal{M}$

We require correctness:

$$\forall(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{Keygen}(1^\lambda), \forall m \in \mathcal{M} : \mathsf{Decrypt}(\mathsf{dk}, \mathsf{Encrypt}(\mathsf{ek}, m)) = m$$

Security of an encryption scheme is typically defined as ciphertexts of different messages being indistinguishable from each other. The minimal requirement for an encryption scheme is called indistinguishability under chosen plaintexts (IND-CPA):

Definition 2.7.2 (IND-CPA security)**.** The advantage of a stateful adversary $\mathcal{A}$ against the IND-CPA property of a PKE scheme PKE is defined as

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{A}}(1^\lambda) = \Pr\left[\begin{array}{l}(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{Keygen}(1^\lambda); b \leftarrow \{0,1\}; (m_0, m_1) \leftarrow \mathcal{A}(1^\lambda, \mathsf{ek}); \\ ct \leftarrow \mathsf{Encrypt}(\mathsf{ek}, m_b); b' \leftarrow \mathcal{A}(ct) : b' = b\end{array}\right] - \frac{1}{2}$$

A scheme PKE is IND-CPA secure if for all stateful PPT adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\lambda)$ so that $\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{A}}(1^\lambda) \leq \mathsf{negl}(\lambda)$

2.7.2. Threshold PKE

A threshold encryption scheme supports secret-sharing the decryption key between multiple parties. Each party can then perform a partial decryption of a ciphertext with its decryption key share, and given enough such partial decryptions, the plaintext can be obtained, without the need to ever reconstruct the decryption key itself.

We follow the notation of [SG02] for threshold encryption:

Definition 2.7.3 (Syntax of Threshold PKE)**.** A (t, n)*-threshold PKE (TPKE)* scheme TPKE consists of a tuple $(\mathsf{Keygen}, \mathsf{Encrypt}, \mathsf{PartDec}, \mathsf{PartVerify}, \mathsf{Combine})$ of PPT algorithms such that

- $\mathsf{TPKE.Keygen}(1^\lambda, n, t)$ outputs a public encryption key ek and sets $\{\mathsf{vk}_i\}_{i\in[n]}$ of verification keys and $\{\mathsf{dk}_i\}_{i\in[n]}$ of decryption key shares.
- $\mathsf{TPKE.Encrypt}(\mathsf{ek}, m)$ takes an encryption key ek, a message m in the message space $\mathcal{M}$, and outputs a ciphertext ct.
- $\mathsf{TPKE.PartDec}(\mathsf{dk}_i, ct)$ takes a decryption key share dk_i, a ciphertext ct, and outputs a decryption share ct_i.
- $\mathsf{TPKE.PartVerify}(ct, \mathsf{vk}_i, ct_i)$ takes a ciphertext ct, a verification key vk_i and a decryption share ct_i, and outputs either 1 or 0. If the output is 1, ct_i is called a *valid* decryption share.
- $\mathsf{TPKE.Combine}(ct, T)$ takes a ciphertext ct and a set of valid decryption shares T such that $|T| = t + 1$, and outputs a message $m \in \mathcal{M}$.

We require correctness, i.e., the following two conditions to hold for all $(\mathsf{ek}, \{\mathsf{vk}_i\}_{i\in[n]}, \{\mathsf{dk}_i\}_{i\in[n]}) \leftarrow \mathsf{TPKE.Keygen}(1^\lambda)$:

1. $\forall m \in \mathcal{M}, ct \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, m), i \in [n]$ it holds that $\mathsf{TPKE.PartVerify}(ct, \mathsf{vk}_i, \mathsf{TPKE.PartDec}(\mathsf{dk}_i, ct)) = 1$.
2. $\forall m \in \mathcal{M}, ct \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, m)$ and any set $T := (ct_1, \ldots, ct_{t+1})$ of valid decryption shares $ct_i \leftarrow \mathsf{TPKE.PartDec}(\mathsf{dk}_i, ct)$ for $t + 1$ distinct decryption key shares dk_i it holds that $\mathsf{TPKE.Combine}(ct, T) = m$.

IND-CPA security can be extended in a straight-forward way to include adversaries that can obtain up to t decryption key shares. While stronger security definitions can be found in the literature, including IND-CCA security which grants adversaries access to a decryption oracle, or threshold versions that allow a combination of decryption key shares and honest partial decryptions, we only require the simple IND-CPA version in this thesis.

We make use of a UC-functionality for distributed key generation $\mathcal{F}_{\text{Keygen}}^{\mathsf{TPKE}}$ defined in Fig. 2.3a.

Functionality $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$

Parameter: Number of parties n, threshold t

- Upon receiving $(\text{Keygen}, 1^\lambda)$ from all parties:
  1. Run $(\text{ek}, \{\text{vk}_i\}_{i\in[n]}, \{\text{dk}_i\}_{i\in[n]}) \leftarrow \text{TPKE.Keygen}(1^\lambda, n, t)$
  2. Send output $(\text{ek}, \text{vk}_i, \text{dk}_i)$ to party P_i

(a) Functionality $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$ for TPKE

Functionality $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$

Parameters: Number of parties n, threshold t

- Upon receiving $(\text{Keygen}, crs_{\text{TSIG}})$ from all parties:
  1. Run $(\text{vk}, \{\text{vk}_i\}_{i\in[n]}, \{\text{sk}_i\}_{i\in[n]}) \leftarrow \text{TSIG.Keygen}(crs_{\text{TSIG}}, n, t)$
  2. Send output $(\text{vk}, \text{vk}_i, \text{sk}_i)$ to party P_i

(b) Functionality $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$ for TSIG

Figure 2.3.: Functionalities for distributed key generation

2.8. Digital Signatures

Digital signatures can be used to verify the authenticity of messages. Similarly to public key encryption, a signature scheme comes with two keys, a secret signing key used to generate signatures, and a public verification key that allows anyone to verify a signature. A valid signature on a message can be used to both identify the sender of the message as well as ensure that the message has not been changed since it was signed.

Definition 2.8.1 (Syntax of Digital Signatures)**.** A *digital signature scheme* SIG over message space $\mathcal{M}$ consists of a tuple $(\text{Keygen}, \text{Sign}, \text{Vf})$ of PPT algorithms such that:

- $\text{SIG.Keygen}(1^\lambda)$ takes 1^λ as input and outputs a keypair (sk, vk).
- $\text{SIG.Sign}(\text{sk}, m)$ takes the signing key sk and a message $m \in \mathcal{M}$ as input and outputs a signature σ.
- $\text{SIG.Vf}(\text{vk}, m, \sigma)$ is deterministic and takes as input a verification key vk, a message $m \in \mathcal{M}$ and a signature σ. It returns either *true* or *false*.

  (Intuitively, returning *true* means that σ is valid signature for message m under verification key vk. Returning *false* means it is invalid.)

SIG is *correct* if for all λ, $(\text{sk}, \text{vk}) \leftarrow \text{SIG.Keygen}(1^\lambda)$, $m \in \mathcal{M}$, and $\sigma \leftarrow \text{SIG.Sign}(\text{sk}, m)$ it holds that

$$\text{SIG.Vf}(\text{vk}, m, \sigma) = \textit{true}.$$

We additionally require a deterministic PPT algorithm $\text{KeyVerify}(\text{sk}, \text{vk})$ so that for $(\text{sk}, \text{vk}) \leftarrow \text{SIG.Keygen}(\lambda)$ it holds that $\text{KeyVerify}(\text{sk}, \text{vk}) = 1$ and if $\text{KeyVerify}(\text{sk}, \text{vk}) = 1$ then $\text{SIG.Vf}(\text{vk}, m, \text{SIG.Sign}(\text{sk}, m)) = \textit{true}$ for all $m \in \mathcal{M}$.

The standard security notion for digital signatures is existential unforgeability under chosen message attacks (EUF-CMA), which requires that given signatures on arbitrary messages, it should not be possible to generate a valid signature on any new message.

Definition 2.8.2 (EUF-CMA-Security [Gol04])**.** A signature scheme SIG = (Keygen, Sign, Vf) is *EUF-CMA-secure* if for every PPT adversary $\mathcal{A}$ with access to a signing oracle O^{SIG} it holds that the following probability is negligible in the security parameter λ:

$$\Pr\left[\begin{array}{l}(\mathsf{sk},\ \mathsf{vk}) \leftarrow \mathsf{SIG.Keygen}(1^\lambda), (\sigma^*,\ m^*) \leftarrow \mathcal{A}^{O^{\mathsf{SIG}}(\mathsf{sk},\cdot)}(\mathsf{vk}, 1^\lambda) : \\ \mathsf{SIG.Vf}(\mathsf{vk},\ m^*,\ \sigma^*) = \mathit{true}\end{array}\right] \leq \mathsf{negl}(\lambda)$$

where σ^* was never returned from $O^{\mathsf{SIG}}(\mathsf{sk}, \cdot)$.

A signature scheme is structure-preserving if its messages, public keys and signatures only consist of elements from $\mathbb{G}_1$ and *grouptwo* and signature verification only needs to evaluate pairing-product equations.

Definition 2.8.3 (Structure-Preserving Signature Schemes)**.** A signature scheme SIG = (Keygen, Sign, Vf) is called structure-preserving with respect to a pairing group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ if the following conditions are all satisfied.

1. The message space $\mathcal{M}$ is $\mathbb{G}_1^m \times \mathbb{G}_2^n$ for some $m, n \in \mathbb{N} \bigcup \{0\}$
2. Public keys and signatures only consist of elements from $\mathbb{G}_1$ and $\mathbb{G}_2$
3. SIG.Vf only requires evaluation of pairing-product equations

2.8.1. Aggregatable Signatures

An *aggregate signature scheme* is a digital signature scheme that additionally allows multiple signatures to be aggregated into a single signature. This single aggregate signature can then be used to verify all the original signatures collectively.

We assume that all signers sign the same message[1] and therefore directly use a syntax to highlight that.

Definition 2.8.4 (Aggregate Signature Scheme for Same Message)**.** We say that an aggregate signature scheme SIG over message space $\mathcal{M}$ consists of the algorithms (Keygen, Sign, Vf) as defined in Definition 2.8.1 as well as the following two additional algorithms:

- $\mathsf{SIG.Agg}((\mathsf{vk}_1, \sigma_1), \ldots, (\mathsf{vk}_\ell, \sigma_\ell))$ takes multiple verification key and signature tuples $(\mathsf{vk}_i, \sigma_i)$ and combines them to a single aggregate signature $\hat{\sigma}$.
- $\mathsf{SIG.AVf}((\mathsf{vk}_1, \ldots, \mathsf{vk}_\ell), m, \hat{\sigma})$ is deterministic and takes as input multiple verification keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_\ell)$, a message $m \in \mathcal{M}$ and a aggregated signature $\hat{\sigma}$. It returns either 0 or 1.

  (Intuitively, returning 1 means that $\hat{\sigma}$ is valid aggregated signature for message m under verification keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_\ell)$. Returning 0 means it is invalid.)

[1] See [BDN18] how to adapt BLS signatures to this setting.

2.8.2. Threshold Signatures

A threshold signature scheme supports secret-sharing the signing key between multiple parties. Each party can then produce a partial signature of a message with its signing key share, and given enough such partial signatures, a full signature on the message can be obtained, without the need to ever reconstruct the signing key itself.

Definition 2.8.5 (Syntax of Threshold Signatures, [Mit+24])**.** A (n, t)-*threshold signature scheme* TSIG over message space $\mathcal{M}$ consists of a tuple (Setup, Keygen, PartSign, PartVerify, Combine, Vf) of PPT algorithms such that:

- $\mathsf{TSIG.Setup}(1^\lambda)$ takes 1^λ as input and outputs a common reference string *crs*.
- $\mathsf{TSIG.Keygen}(crs, n, t)$ takes the common reference string along with integers n and t s.t. $1 \leq t \leq n$ as input and outputs signing/verification keys $(\mathsf{sk}_i, \mathsf{vk}_i)$ along with a global verification key vk.
- $\mathsf{TSIG.PartSign}(crs, \mathsf{sk}_i, m)$ takes the common reference string, the i-th signing key and a message vector $m \in \mathcal{M}$ as inputs and outputs a partial signature σ_i.
- $\mathsf{TSIG.PartVerify}(crs, \mathsf{vk}_i, m, \sigma_i)$ takes the crs, the i-th verification key, a message $m \in \mathcal{M}$ and a partial signature σ_i as input and outputs 1 if the partial signature is valid and 0 otherwise.
- $\mathsf{TSIG.Combine}(crs, T)$ takes the crs and a set of valid partial signatures T with $|T| = t + 1$ and outputs a signature σ.
- $\mathsf{TSIG.Vf}(crs, \mathsf{vk}, m, \sigma)$ takes the crs, a verification key, a message $m \in \mathcal{M}$ and a signature and outputs 1 if the signature is valid and 0 otherwise.

TSIG is *correct* if for all λ, $crs \leftarrow \mathsf{TSIG.Setup}(1^\lambda)$, $(\{\mathsf{sk}_i, \mathsf{vk}_i\}, \mathsf{vk}) \leftarrow \mathsf{TSIG.Keygen}(crs, n, t)$, $m \in \mathcal{M}$, $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs, \mathsf{sk}_i, m)$ for $i \in [1, n]$, $\forall T \subset \{\sigma_i\} : |T| = t + 1$ and $\sigma \leftarrow \mathsf{TSIG.Combine}(crs, T)$ it holds that $\mathsf{TSIG.Vf}(crs, \mathsf{vk}, m, \sigma) = 1$

Similar to EUF-CMA-security for normal signatures, we require TSIG to be threshold unforgeable.

Definition 2.8.6 (Threshold Unforgeability [Mit+24])**.** Let TSIG = (Setup, Keygen, PartSign, PartVerify, Combine, Vf) be an (n, t)-threshold signature scheme over message space $\mathcal{M}$. TSIG is called *TS-UF-1 secure* if for any PPT adversary $\mathcal{A}$ the advantage in the security game $\mathsf{Exp}^{TS-UF-1}_{\mathcal{A}}(1^\lambda)$ defined in Fig. 2.4 is negligible.

As with threshold encryption, we make use of a UC-functionality for distributed key generation $\mathcal{F}^{\mathsf{TSIG}}_{\mathrm{Keygen}}$ defined in Fig. 2.3b.

$\mathsf{Exp}^{\mathsf{TS-UF-1}}_{\mathcal{A}}(1^\lambda)$

1 : $crs \leftarrow \mathsf{Setup}(1^\lambda)$
2 : $(n, t, \mathrm{CS}, \mathrm{st}_0) \leftarrow \mathcal{A}(crs)$
3 : $\mathrm{HS} \coloneqq [1, n] \setminus \mathrm{CS}$
4 : $(\{\mathsf{sk}_i, \mathsf{vk}_i\}, \mathsf{vk}) \leftarrow \mathsf{Keygen}(crs, n, t)$
5 : $(m^*, \sigma^*, \mathrm{st}_1) \leftarrow \mathcal{A}^{\mathrm{O}^{\mathsf{PartSign}}}(\mathrm{st}_0, \mathsf{vk}, \{\mathsf{sk}_j\}_{j\in\mathrm{CS}}, \{\mathsf{vk}_i\}_{i\in[1,n]})$
6 : **return** $(\mathsf{Vf}(crs, \mathsf{vk}, m^*, \sigma^*) \wedge |\mathrm{CS}| \leq t \wedge |S_1(m^*)| \leq t - |\mathrm{CS}|)$

Oracle $\mathrm{O}^{\mathsf{PartSign}}(i, m)$

1 : **assert** $m \in \mathcal{M} \wedge i \in \mathrm{HS}$
2 : $\sigma_i \leftarrow \mathsf{PartSign}(crs, \mathsf{sk}_i, m)$
3 : **if** $\sigma_i \neq \perp$
4 : $\quad S_1(m) = S_1(m) \cup \{i\}$
5 : **return** σ_i

Figure 2.4.: Security game defining TS-UF-1 security for threshold signatures.

2.9. Commitments

Commitments enable one party to commit itself on the content of some message towards one or more other parties while keeping the content hidden.

Definition 2.9.1 ((Non-interactive) Commitment Scheme)**.** A non-interactive commitment scheme COM with message space $\mathcal{M}$ consists of a tuple (Setup, commit, open) of PPT algorithms such that

- $\mathsf{Setup}(1^\lambda)$ outputs a common reference string *crs*
- $\mathsf{commit}(crs, msg)$ takes a common reference string and a message and outputs a commitment *com* and opening information *decom*
- $\mathsf{open}(crs, com, decom, msg)$ takes a common reference string, a commitment *com*, opening information *decom* and message *msg* and outputs *true* if *com* is a valid commitment on *msg*

We require correctness:

$$\forall msg \in \mathcal{M}, crs \leftarrow \mathsf{Setup}(1^\lambda) :$$
$$(com, decom) \leftarrow \mathsf{commit}(crs, msg) \Rightarrow \mathsf{open}(crs, com, decom, msg) = true$$

We will often leave out the crs if it is clear from the context.

Security of commitment schemes is defined through their *hiding* and *binding* properties. Hiding ensures that no information about the message can be learned from a commitment, while binding ensures that the committer can only open the commitment to one value.

Definition 2.9.2 (Hiding property)**.** A commitment scheme COM is called hiding if there exists a negligible function $\mathsf{negl}(\lambda)$ so that for all PPT adversaries $\mathcal{A}$ it holds that the advantage $\mathsf{Adv}^{\mathsf{hiding}}_{\mathcal{A}}(1^\lambda) \leq \mathsf{negl}(\lambda)$, where the advantage is defined as:

$$\mathsf{Adv}^{\mathsf{hiding}}_{\mathcal{A}}(1^\lambda) = \Pr\left[\begin{array}{l} crs \leftarrow \mathsf{Setup}(1^\lambda); b \leftarrow \{0,1\}; \mathcal{M} \ni (m_0, m_1) \leftarrow \mathcal{A}(1^\lambda, crs); \\ (com, decom) \leftarrow \mathsf{commit}(crs, m_b); b' \leftarrow \mathcal{A}(com) : b' = b \end{array}\right] - \frac{1}{2}$$

Definition 2.9.3 (Binding property)**.** A commitment scheme COM is called binding if there exists a negligible function $\mathsf{negl}(\lambda)$ so that for all PPT adversaries $\mathcal{A}$ it holds that the advantage $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{binding}}(1^\lambda) \leq \mathsf{negl}(\lambda)$, where the advantage is defined as:

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{binding}}(1^\lambda) = \Pr\left[\begin{array}{l} crs \leftarrow \mathsf{Setup}(1^\lambda); (com,\ m_0,\ decom_0,\ m_1,\ decom_1) \leftarrow \mathcal{A}(1^\lambda,\ crs) : \\ \mathsf{open}(crs,\ com,\ decom_0,\ m_0) = true \wedge \\ \mathsf{open}(crs,\ com,\ decom_1,\ m_1) = true \wedge m_0 \neq m_1 \end{array}\right]$$

Commitment schemes can additionally have homomorphic properties. In this thesis, we make use of additively homomorphic commitments.

Definition 2.9.4 (Additively homomorphic commitment scheme)**.** Let the message space $\mathcal{M}$ be an additive group $(\mathcal{M}, +)$. A commitment scheme COM is called additively homomorphic if it comes with two additional algorithms $(\mathsf{add\_c},\ \mathsf{add\_d})$ with the property that:

$$\begin{array}{l} \forall m_0, m_1 \in \mathcal{M}, crs \leftarrow \mathsf{Setup}(1^\lambda), \\ (com_0,\ decom_0) \leftarrow \mathsf{commit}(crs,\ m_0), (com_1,\ decom_1) \leftarrow \mathsf{commit}(crs,\ m_1) : \\ \mathsf{open}(crs,\ \mathsf{add\_c}(com_0,\ com_1),\ \mathsf{add\_d}(decom_0,\ decom_1),\ m_0 + m_1) = true \end{array}$$

We will write $com_0 \oplus com_1$ as shorthand for $\mathsf{add\_c}(com_0,\ com_1)$ and $decom_0 \oplus decom_1$ as shorthand for $\mathsf{add\_d}(decom_0,\ decom_1)$.

Structure Preserving Commitment Schemes As already mentioned in Section 2.5, an important property of building blocks which we want to use in zero-knowledge proofs is that they are structure-preserving.

Definition 2.9.5 (Structure-Preserving Commitment)**.** A commitment scheme COM = (Setup, commit, open) is called structure-preserving with respect to a pairing group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ if the following conditions are all satisfied.

1. Common parameter gp consists of a group description gp generated by G and constants $a_{ij} \in \mathbb{Z}_p$.
2. Commitment and unveil messages consist of group elements in $\mathbb{G}_1$ and $\mathbb{G}_2$.
3. Opening algorithm open consists only of evaluating membership in $\mathbb{G}_1$ and $\mathbb{G}_2$ and relations described by pairing product equations.

3. PUBA: Storing data on the user device

This chapter is based on "PUBA: Privacy-Preserving User-Data Bookkeeping and Analytics" [Fet+22b; Fet+21] and most parts of this chapter are rephrased from these works. However, both in the journal version [Fet+22b] as well as the eprint version [Fet+21], large parts of the formal description of the ideal functionality and protocol are relegated to the appendix. In this thesis, the content underwent significant reorganization and was adjusted to make the notation consistent.

My Contribution In the paper, I equally contributed to the formal modelling, the protocol design and security proofs, with a special focus on practical instantiability. I implemented and benchmarked the prototype.

Organization of this Chapter We first provide an introduction to privacy-preserving data collection and the related work in Section 3.1. Then we discuss the desired functionality and involved parties in Section 3.2. In Section 3.3 we present $\mathcal{F}_{\text{PUBA}}$ to formalize the desired security properties. Section 3.4 then contains the description of the protocol Π_{PUBA}. Details on how to use PUBA in two example applications are given in Section 3.5. Finally, Section 3.6 contains our prototype implementation and benchmark results.

3.1. Introduction

Privacy-enhancing cryptographic protocols could be highly beneficial to citizens and our society in a multitude of user-centric scenarios including mobile payments, loyalty systems, ticketing systems, toll collection, web search, participatory sensing, disease prediction and pay-as-you-drive insurance. Also, operators of such systems should be encouraged to deploy strong privacy-enhancing technologies in view of EU's GDPR and the severe fines in case of violation.

However, despite their advantages, proposed cryptographic protocols to protect privacy in such scenarios are rarely in real-world use today. The reasons for this are manifold; aside from development and deployment costs, this is caused by a gap between what these protocols offer and what is required when targeting real-world applications: While academic proposals typically strive for "perfect" anonymity and a simple functionality,

they neglect scenario-specific real-world requirements stemming from business models, laws, regulations, or user needs. These requirements often conflict with the desire for perfect anonymity. For instance, fraud detection and anti-money laundering as mandated for digital payments or targeted advertising as performed in loyalty systems are based on a customer's transaction history. Neglecting these types of requirements when building privacy-preserving systems, e.g., an anonymous mobile payment or loyalty system, hinders practical deployment. In this chapter, we take an important step towards resolving this issue by proposing a flexible framework that enables the privacy- and authenticity-preserving bookkeeping and analytics of user history data. Our building block is secure in the Universal Composability (UC) framework and can thus be securely incorporated into privacy-preserving systems, e.g., anonymous mobile payments, to satisfy requirements relying on the analysis of participating users' behavioral data, such as fraud detection.

For concreteness, we consider two examples in detail: loyalty systems and mobile payments. In loyalty programs such as Optimum [Lob20] in Canada, Payback [PAY20] in Germany, or Nectar [Aim20] in the UK, customers collect loyalty points (e.g., using their smartphones) for purchases in participating shops which they can redeem at a later point to pay for purchases, get vouchers, etc. Academic proposals like black-box accumulators (BBAs) [Har+17] exist, which realize this point collection and redemption functionality in a privacy-preserving way while protecting the operator from forgeries and double-redemption. However, BBAs do not account for the business models of loyalty system providers, which typically consist of earning money by enabling advertising partners to place targeted ads and coupons. To this end, the operator keeps track of users' purchase histories. Ad selection is typically performed using a machine-learning-based classifier which maps purchase histories to product categories customers might find interesting. Advertisers promoting their products send ads for certain product categories to the operator along with the price they are willing to pay every time their ad is delivered by email or displayed on the loyalty program app. Users are additionally motivated to participate in the targeted advertisement service by the opportunity to obtain incentives, such as targeted coupons offering discounts on certain products. In currently deployed systems, points-of-sale report purchases to the operator along with a customer's ID. This way, authentic and up-to-date purchase histories can be kept for each customer. Since the ad selection procedure is executed in the background on the operator's side, correctness of this computation is ensured. Conversely, a user's purchases are linked, and the operator—or anyone else gaining access to this history—may misuse it to obtain additional knowledge about the user's locations and behavior.

NFC-based mobile payments are an increasingly popular method to conduct payments for goods or services at points-of-sale. This payment mode has been boosted with the recent roll-out of Google Pay and Apple Pay. While deployed systems essentially realize a virtual credit or debit card (making payments linkable), privacy-preserving systems could theoretically be built from anonymous e-cash, e.g., see [AB09]. However, these constructions do not account for current regulations for mobile payments, such as fraud detection as required by the 2nd European Payment Services Directive (PSD2) [PC15]. Fraudulent transactions can occur due to lost or stolen payment devices or compromised payment credentials stored on them. Fraud detection involves continuously monitoring

a customer's transaction history for anomalies and typical fraud patterns. This can be done using a set of rules or sophisticated machine learning algorithms. Based on these background checks, a decision is made in real time regarding whether a newly initiated transaction should be granted or denied.

To realize the two scenarios described above—as well as many others—in a privacy-preserving manner, a new building block is needed that ensures privacy-preserving bookkeeping as well as analytics of user history data: A user's actions generate data (e.g., transaction value, date/time, location, type of shop, etc.) observable by an operator. A logbook kept solely on the user's device should be updated with the observed data such that its content cannot be manipulated by the user while ensuring its current content remains hidden from the operator. The former is particularly important for requirements like fraud or money laundering detection. More precisely, we need to ensure the authenticity, integrity, freshness, confidentiality, and unlinkability of the logbook. At the same time, the building block needs to allow for analytical computations, where users are incentivized or required to take part in privacy-preserving analytics of their latest logbook. These computations must not leak the input(s) and the operator may only obtain privacy-preserving statistics, classification, etc. as output. The user might also receive output, such as targeted ads in a loyalty system. Furthermore, in some cases, a minor update to the logbook with results from the analytics computation might be required, e.g., a risk level resulting from the background fraud detection checks. Depending on the scenario, the analytics function itself might need to remain confidential, e.g., when considered intellectual property (such as a classifier for targeted advertising). In this case, it needs to be ensured that the operator cannot misuse this opportunity to selectively deploy privacy-violating functions.

We distinguish two types of analytics computations: outsourced heavyweight analytics and direct lightweight analytics. As the privacy-preserving evaluation of machine-learning classifiers (e.g., to determine a user's individual fraud risk level) might be computationally and communication-wise very demanding, in particular when mobile user devices are involved, it must be possible to securely outsource this computation to a proxy server that acts on behalf of the user(s). Ideally, even a malicious proxy should learn nothing about the user and operator inputs or the output provided to the user. In some scenarios, outsourced analytics should not block further interactions with the system, i.e., users should be able to continue updating their logbooks while analytics are running. For instance, in loyalty systems customers should be able to continue shopping and keep track of purchased items while the outsourced ad selection process (for a previous version of the logbook) is still in progress. Direct lightweight privacy-preserving analytics are performed jointly by the user's device and the operator. This type of analytics might be done prior to an update of the logbook to determine further actions, e.g., whether the payment transaction just initiated should be accepted in view of the customer's personal risk level.

3.1.1. Our Contribution

In this chapter, we focus on properly formalizing and instantiating the functionality and security and privacy properties of a bookkeeping and analytics mechanism as sketched in the previous section. To this end, we present PUBA, a framework for privacy-preserving user-data bookkeeping and analysis.

While numerous practical application scenarios would evidently benefit from privacy-preserving bookkeeping and analytics, to the best of our knowledge, this is the *first* work to formally study such a multi-purpose building block.

Our contribution consists of a formal UC-based modeling, protocols for managing the logbook along with security proofs, implementations and benchmarks of crucial protocols, as well as exemplary instantiations of fraud detection and targeted advertising in our framework. This work is one of the few that combines a complex yet practical cryptosystem with a thorough UC security analysis. Details are provided below.

Security model Based on a high-level system architecture (see Section 3.2) including a set of intuitive but informal security properties (see Section 3.2.5), we designed a flexible ideal functionality (see Section 3.3) in the UC framework [Can00; Can01]. Providing a model and a security analysis in the UC framework is essential as we expect PUBA to be used as a building block for larger privacy-preserving systems. A primary challenge lays in finding an appropriate trade-off between flexibility (to cover a broad set of applications) and complexity. Our functionality covers users, an operator, a trusted signing authority (verifying the privacy-friendliness of analytical functions), and proxies interacting in preparatory tasks like user registration and validation of analytics functions, as well as operational tasks such as bookkeeping (which also covers direct analytics), outsource, analytics, and update. The correct computation of an application-specific function Δ contained in both Bookkeeping and Analytics is managed by a subfunctionality $\mathcal{F}_{\text{PPA}}$ which provides independence of the used method of computation. The subfunctionality can be instantiated, for example, with any secret-sharing-based general MPC framework (for example based on the SPDZ [Dam+12] family) that treats Δ as a black-box, which allows for flexible instantiations and lets PUBA easily profit from performance advances in MPC. It is, however, also possible to provide non-black-box instantiations of $\mathcal{F}_{\text{PPA}}$ that exploit the structure of Δ, e.g., by letting the user prove in zero-knowledge that the function was computed correctly. This enables more efficient computations, which is particularly desirable for the Direct Analytics included in the Bookkeeping task.

Provably secure protocols We designed cryptographic protocols for the tasks mentioned above in Section 3.4.8 and provide a full security proof for our construction in Section A.3. Our main challenge was to carefully combine different primitives and techniques into protocols that are both secure in the UC framework and efficient at the same time. To this end, we follow a commit-sign-rerandomize-prove approach to achieve

the desired security properties for our logbook. We build on Groth-Sahai [GS08] non-interactive zero-knowledge proofs, which are fairly practical and secure under standard assumptions (e.g., SXDH over bilinear groups). Furthermore, we use structure-preserving signatures [Abe+11] and homomorphic multi-commitments [Abe+15] for which statements (e.g., knowledge of a signature on a commitment) can be efficiently proven using Groth-Sahai.

Implementation and benchmarks We implemented our protocols for user registration (UReg), Bookkeeping, Outsource and Update as well as an exemplary Analytics computation based on logistic regression using the MP-SPDZ MPC framework [Kel20]. For the user side, we used Nexus 5X and Galaxy S8 smartphones, whereas operator and proxy computations were implemented on more powerful server hardware. Our measurements show that even fairly complex Bookkeeping involving permuting, setting, and adding values to entries can be completed in approximately 2 seconds for a logbook containing 100 entries and assuming an NFC data rate of 424 kbit/s. Outsourcing the logbook for an analytics computation takes around 660 ms, assuming a mobile data rate of 10 Mbit/s. Retrieving the results of the analytics and (potentially) updating the logbook takes 2 seconds, assuming the same data rate. The privacy-preserving analytical computation itself takes around 380 ms for co-located operator and proxy servers. These estimates demonstrate that privacy-preserving bookkeeping and analytics can indeed be practical. Our implementation is available at `https://github.com/kastel-security/puba`.

Application: Fraud detection We sketch how a simple two-tier fraud detection system could be realized with PUBA. Tier 1 enforces the user, within the scope of an Analytics computation, to perform a more complex machine-learning-based fraud detection with the operator once a certain number of payment transactions is reached. This results in some risk level which is stored in the logbook. Tier 2 is a Direct Analytics computation performed when a new payment transaction is initiated. It consists of checking simple rules that take the risk level into account to decide whether to accept or decline the transaction. Assuming the hardware and data rates described above and a logbook storing the last 20 transaction records (each consisting of 5 values), we estimate Tier 1 detection using logistic regression to take around 3 seconds (including Outsourcing and Update), while Tier 2 detection including Bookkeeping can be completed in less than 2 seconds.

3.1.2. Related Work

To the best of our knowledge, this work is the first to formally define a multi-purpose privacy-preserving bookkeeping and analytics building block.

Regarding bookkeeping, we use mechanisms similar to the Black-Box Accumulators (BBA) introduced in [JR16; Har+17]. BBA provides a framework for *privacy-preserving point collection and redemption* but does not consider data analytics. Moreover, its security is defined using a game-based approach, which leads to weaker security guarantees.

In [Blö+19], updatable anonymous credentials are introduced, allowing the dynamic modification of a credential's attributes via an update function after creation. However, the updatable credentials defined in [Blö+19] do not satisfy our requirements. The input to the update function is provided solely by and known only to the user. Moreover, credential freshness and their use for analytical computations are not considered. Like [JR16; Har+17], they do not consider security within the UC model.

Kolesnikov, Kumaresan, and Shikfa [KKS12] present schemes providing *input authentication* in the following sense: in their model, malicious clients perform secure multi-party computations with a semi-honest server, and the scheme ensures that clients use the *same input* across multiple computations. This can be considered a weak form of bookkeeping with less stringent requirements. While [KKS12] ensures that users cannot use arbitrary inputs for computations, the users can be linked across interactions. Furthermore, in contrast to [KKS12], our framework also works with malicious servers and provides additional security guarantees for honest users that are not directly inherited from the MPC.

Privacy-preserving analytics and data mining (without bookkeeping and input authentication) have been well-studied. Proposed approaches in the literature either construct dedicated protocols exploiting the structure of existing methods (such as neural networks or logistic regression) commonly used for machine learning [Ash+17; Cet+17], or propose techniques for efficiently solving specific analytical problems [Yu+18; Sen+05; WR10; DZ16].

Let us briefly consider related work regarding the applications of PUBA studied in this chapter: *targeted advertising in loyalty systems* and *fraud detection in mobile payments*. To the best of our knowledge, privacy-preserving targeted advertising for loyalty programs has not been thoroughly analyzed before. The majority of work on privacy-preserving targeted advertising deals with an Internet scenario where an advertisement network (such as Google) wants to display targeted advertisements on websites based on the user's prior web activities. With a few exceptions, e.g., [Bac+12], the security and privacy properties of proposed systems are not formally proven. Regarding ad selection, there are currently two prevalent approaches: client-based and proxy-based selection. In client-based ad selection (e.g., [Tou+10; Bac+12; GLM16]), a browser plugin typically keeps track of the user's online behavior and interests and classifies the user to determine the most relevant ad. This implies that the ad network's selection algorithm must be public, and the correctness of inputs and computation is not ensured. In proxy-based selection (e.g., [GCF11]), a trusted third party (TTP) with secure channels to the user hides user data from the ad network. This implies a dependency on the TTP and assumes that users send correct data.

Little work has been devoted to privacy-preserving fraud detection for payments. The most relevant work, [Can+18], performs an exploratory study on the performance of fully homomorphic encryption-based classification of finance-related transaction data run by a fraud detection service provider. Authenticity and freshness of the data is assumed.

3.2. High Level Description

In this section, we provide a high-level introduction to the parties and tasks involved in the analytics framework for PUBA.

3.2.1. Parties and Roles

PUBA considers four different types of parties:

Users U PUBA allows for arbitrarily many users to participate in the system. Each user collects data inside a personal logbook Log, which they can provide for privacy-preserving computations. We call the main part of this data the *user history* $\mathcal{UH}$. This part is authenticated and cannot be changed by the user at will, but is only updated through private bookkeeping tasks with the operator. The private data is represented by a *vector* of *slots*, where each slot contains a $\mathbb{Z}_p$ element. Additionally, the logbook contains some data needed to ensure correct and private operation of the system. Users may be corrupted and maliciously collude with other corrupted parties. We assume that users participate using relatively weak hardware. For many real-world applications, the user side of PUBA will likely be realized as a smartphone application.

The operator O For any given instance of PUBA, there is exactly one operator. This party is the central entity managing the system. The operator has an interest in evaluating analytical functions on the data collected by users. In certain scenarios, the precise details of these functions, however, may be subject to trade secrets and should not be leaked to any other party. We thus assume that the operator can *hide* all sensitive information inside function parameters (FPs) *fp* while the publicly known function Δ itself only provides the general structure. In real-world applications, FPs may correspond to transition matrices of neural networks [JVC18] or weights and features for linear regression. As the operator's input to analytics computations, the FPs achieve the same level of privacy as the users' private data. Relying on FPs is without loss of generality [Val76] with respect to the class of computable functions. The operator may be corrupted and maliciously collude with other corrupted parties, apart from proxies (which does not affect user privacy).

The trusted signing authority $\mathcal{TSA}$ Allowing arbitrary private FPs precludes any meaningful level of privacy for the users. To prevent malicious operators from using FPs which trivially undermine the users' privacy, we assume the existence of a trusted signing authority (TSA): Before using FPs for computations, the operator must have them certified by the TSA to ensure they do not violate privacy requirements. We elaborate on the difficulties of this task in Section A.1.1. PUBA enforces that only FPs certified by the TSA can be used during computations on the users' data. The TSA is an external entity and has to be trusted by users as well as the operator. We envision the TSA as a

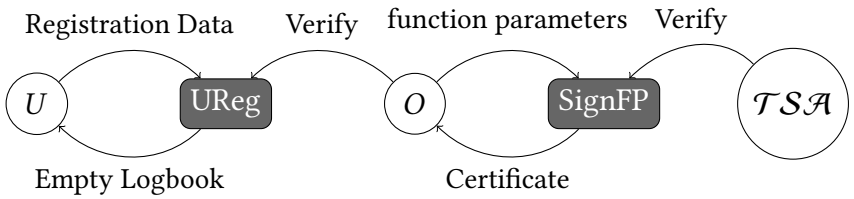

Figure 3.1.: Overview of the preparatory tasks.

privacy-defending, donation-funded NGO (like the Electronic Frontier Foundation) or a governmental body like the Federal Data Protection Officer.

Proxies P PUBA allows arbitrarily many proxies to participate. Proxies are used for computing analytical functions and to coordinate computations that require data from multiple users. We assume proxies provide the computational power and bandwidth of a regular server. Proxies may be corrupted and collude with corrupted users. As long as proxies do not collude with the operator, PUBA guarantees that all user data remains hidden from both the proxy and the operator. Any user may potentially set up their own proxy, yet it is also possible that NGOs provide donation-funded proxy servers.

3.2.2. Preparatory Tasks

Before users can participate in the system, they have to register with the operator. Similarly, the operator must have function parameters signed by the TSA before using them. Both tasks have to be conducted only once (per user/FP respectively) before computations with them are possible. An overview of these tasks is provided in Fig. 3.1.

SignFP Any function computed with PUBA is split in two parts: a generic and publicly known function representation Δ (e.g., a neural network with padded neurons) and a corresponding set of function parameters which determine the specifics (e.g., the transition matrices). These function parameters constitute sensitive data held by the operator. Before the operator can use any function parameters for computations, the TSA has to verify that the resulting function does not violate the required privacy standards. To that end, the TSA verifies the FPs according to (but not limited to) the following criteria: 1. Is the output sufficiently general to not leak confidential user information? We assume a public catalog of requirements FPs have to fulfill. This also provides feedback for the users on the expected level of privacy. 2. Does this function use the slots of the user history according to their specification? We generally assume the specification of the user history—the semantic interpretation of the individual slots—to be public knowledge. 3. Are there any additional leaks when combining these FPs with any of the previously certified FPs? We model the verification function such that the TSA can input all previously accepted FPs as auxiliary input. The actual SignFP task is depicted in Fig. 3.1: The operator inputs FPs

which are then checked by the TSA against the privacy guidelines. If the FPs comply with the guidelines, the TSA provides the operator with some form of certificate. Our protocol uses signatures on the FPs. This certificate is a required input to any computation task that uses these FPs.

UReg A user stores their private user history inside a personal logbook Log. Providing the user with an "empty" logbook which contains an authenticated initial user history is the purpose of the user registration task. The task is depicted in Fig. 3.1 and consists of three phases. In the first phase, the user identifies themselves. This triggers the second phase, in which the operator verifies that the user is not already registered—we require that each user can only have *at most* one logbook. Lastly, both parties jointly compute the initial logbook containing an authenticated user history. For most scenarios, the user history is supposed to start out empty. For scenarios where the initial user history should contain some specific data, this task allows its computation based on appropriate FPs. The logbook Log is a requirement for participation in any of the other tasks. The UReg task is the *only* task identifying the user and is performed only once.

3.2.3. Bookkeeping

A central focus of PUBA is privacy-preserving bookkeeping, which allows the operator and a user to manipulate data with certain guarantees for both parties.

Bookkeeping The Bookkeeping task manipulates a single user's data. While the main purpose of this task is to provide an efficient targeted manipulation of the user history in an authenticated yet privacy-preserving way (such as adding or resetting individual values), the task optionally performs *lightweight* direct analytics on the user data according to an application-specific function Δ and the certified FPs. This enables manipulations of the user history based on its current values, which may be required for scenarios such as fraud detection, where the decision (to be recorded) of whether a transaction is granted depends on a risk level stored in the user history. The main part—authenticated manipulation of the user history—is done without leaking the authenticated and private data stored in the user history to the operator. The user history is updated using update information $\mathcal{UI} = (\alpha, \mathbf{s}, \mathbf{a})$. These contain three consecutive operations defined by three maps output by Δ during the computation. The variable α contains a permutation, which is applied to the user history to create a temporary user history as

$$\mathcal{UH}' \leftarrow \alpha(\mathcal{UH}).$$

This permutation changes the slots of some (or all) entries, while not changing the contents itself. For example, $\mathcal{UH} \coloneqq (9,\ 8,\ 7)$ and $\alpha \coloneqq \left(\begin{smallmatrix} 0 & 1 & 2 \\ 2 & 1 & 0 \end{smallmatrix}\right)$[1] would yield $\mathcal{UH}' \coloneqq (7,\ 8,\ 9)$. After

[1] Note that since our user history slots are numbered from 0 to $m-1$, we also define the permutation that way.

the permutation, a set operation is executed. The set vector $\mathbf{s}$ has length m and contains either $\mathbb{Z}_p$ elements or $\perp$. A new temporary user history is then created via

$$\mathcal{UH}'' \coloneqq (uh''_0, \ldots, uh''_{m-1})$$
$$\text{with } uh''_i \coloneqq \begin{cases} \mathbf{s}[i] & \text{for } \mathbf{s}[i] \neq \perp \\ \mathcal{UH}'[i] & \text{for } \mathbf{s}[i] = \perp \end{cases}.$$

To continue our small example, $\mathcal{UH}' \coloneqq (7,\ 8,\ 9)$ and $\mathbf{s} \coloneqq (1,\ 3,\ \perp)$ would yield $\mathcal{UH}'' \coloneqq (1,\ 3,\ 9)$. Finally, an add operation is performed by simply adding the vector $\mathbf{a}$ (which also has length m) to $\mathcal{UH}''$ to create the final new user history:

$$\mathcal{UH}^{\text{new}} \leftarrow \mathcal{UH}'' + \mathbf{a}.$$

Finishing the example, $\mathcal{UH}'' \coloneqq (1,\ 3,\ 9)$ and $\mathbf{a} \coloneqq (4,\ 0,\ 0)$ would yield $\mathcal{UH}^{\text{new}} \coloneqq (5,\ 3,\ 9)$.

While the additive increment is hidden from the operator, the permutation and direct update are learned by both parties and hence subject to privacy considerations during auditing with the TSA to ensure that they leak no personal information on the user. Of course, if permutations and direct updates are not needed, the user can skip them in favor of more efficient updates using only the additive increment. The chosen three-operation update mechanism is a tradeoff between what is usually required for (basic) bookkeeping applications and what can be efficiently implemented (using zero-knowledge proofs and homomorphic commitments).

The user history remains authenticated because 1. the operator knows that the input was authenticated, and 2. the correctness properties ensure that the data was manipulated correctly.

3.2.4. Outsourcing Analytical Computations

We assume users have relatively weak hardware incapable of securely computing analytical functions like neural networks efficiently. To enable these costly computations, we involve an additional party—the proxy—which provides the computational power and bandwidth required to perform such computations. As long as the proxy does not collude with the operator it learns no secrets or even analytical results from participation. A corrupted proxy also poses no risk for the operator's privacy. Involving a proxy enables *synchronization* of analytical tasks which require private data from several users. The basic workflow for outsourcing analytical tasks is shown in Fig. 3.2; note that the three different tasks involved in this cannot be scheduled arbitrarily but must be conducted sequentially. The general workflow first lets the user *distribute* the logbook containing their current user history to a proxy of their choice and to the operator using the Outsource task. This process will disable outsourcing the logbook until this chain of Outsource-Analytics-Update tasks is completed. These shares are then used by proxy and operator for computing the analytical function using the Analytics task. This can potentially cause updates on the user history which

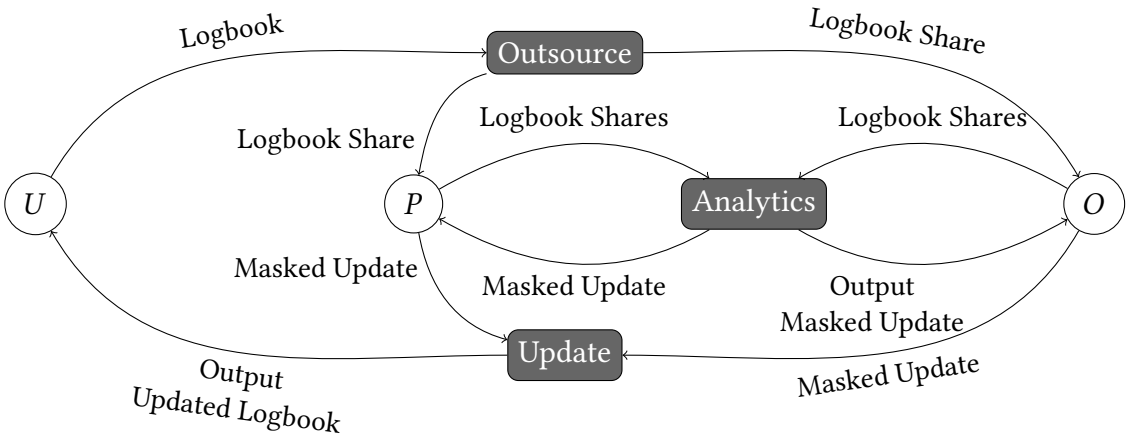

Figure 3.2.: Overview of tasks for outsourcing computations.

follow the same three basic operations (permutation, direct update, additive increment) that were also used for the Bookkeeping task. Again, we want to hide the additive vector applied to the user history and thus use a similar mechanism as during the Bookkeeping task, except that the values relevant for the user are temporarily stored by the proxy and hence masked with one-time pads. The update can later be applied by the user by means of the Update task which also re-enables the Outsource task for that user.

Outsource To prepare the computation of an analytical task the Outsource task is conducted between user, proxy and operator as shown in Fig. 3.2: The user inputs their current logbook, which contains the latest authenticated user history. A snapshot of this is shared between the operator and proxy for use in the computation, while the user receives a new logbook identical to the old one, except that it is now marked as having been outsourced.

Analytics Using the shares obtained from the Outsource task, the proxy and operator can now conduct the actual analytics computation. This is depicted in Fig. 3.2 as well. The function is computed on the private user data using any secret-sharing based MPC framework. The operator learns the desired analytical result directly. Private analytical output for the user is masked and output to the proxy; the users fetch the data and reconstruct it at their convenience (using the Update task), but the proxy does not learn the data. Computation again follows the function Δ and requires validated FPs from the operator.

Update As mentioned before, the results of Analytics can be used to update the user data. To apply these to the latest user history, PUBA provides the Update task (see Fig. 3.2). This also re-enables the Outsource task for the user's logbook. The masked analytical output destined for the user is forwarded to them. Both the operator and proxy input

the share they obtained from the Analytics task. These shares contain the updates for the user history, the computation results relevant for the user and additional information to detect tampering of the shares. The user inputs the latest user history and obtains a new authenticated user history updated using the same mechanism as in the Bookkeeping task.

The triplet from Fig. 3.2 is *non-blocking* regarding further Bookkeeping tasks: A user can outsource the latest user history in an Outsource task, run analytics in the "background", and then update the user history using the Bookkeeping task arbitrarily often *before* participating in the Update task. As mentioned, this is desired, e.g., in scenarios like targeted advertising. We stress, however, that the triplet is blocking with respect to further Analytics tasks (amongst others, to hamper denial of service attacks on proxies and operator): After executing the Outsource task, the user has to successfully perform the Update task before they can call Outsource again. This implies that the user history input to the Update task has potentially been changed via successive Bookkeeping tasks and may significantly differ from the user history used during Outsource. Special care must be taken during function design to ensure that the update remains meaningful even after any number of Bookkeeping tasks have manipulated the user history in the meantime. This can, e.g., be done by defining a separate portion of the user history which can only be modified by Update.

3.2.5. Security Guarantees

This section discusses the security requirements of PUBA in an informal fashion. The ideal functionality $\mathcal{F}_{\text{PUBA}}$ in Section 3.3 is based on these. We explain how the protocol achieves these properties in Section 3.4.5.

The *operator* expects authenticated inputs and correct analytical results, while keeping potential trade secrets hidden. In particular, we desire the following security requirements for the operator:

Owner Binding Users can only use *their own* user history. Even corrupted users cannot steal an honest user's user history.

History Freshness No user can use an outdated user history for any of the Bookkeeping, Outsource, or Update tasks: The same user history can never be used twice without the operator noticing.

History Unforgeability The user history can only be changed through task executions. It is computationally infeasible for a user to create a user history with arbitrary values that would be accepted by the operator. This hampers *Model Extraction Attacks* [Tra+16] in which the user uses targeted inputs to steal the FPs.

Uniqueness Each user can have *at most* a single logbook. It is not possible for a user to own two valid logbooks at the same time.

Function Privacy The function parameters input by the operator remain private: Only the TSA is allowed to learn them. Other parties only learn the output of the function computed with these FPs.

Any *user* interacting with PUBA expects privacy of the collected data throughout different interactions:

Unlinkability For multiple interactions through the Bookkeeping protocol or the Outsource/Analytics/Update process, it should not be possible to determine which user they belong to, or even whether they belong to the same or different users. Users can only be identified during registration.

Input Privacy The system does not reveal anything about the user history that cannot be derived from the result of the computation.

Function Parameter Binding Computations can *only* be performed using FPs that were previously certified by the TSA. It is not efficiently possible for the operator to use uncertified FPs.

3.3. The Ideal Functionality $\mathcal{F}_{\text{PUBA}}$

Having presented the system informally with its security requirements, we now provide the complete formal specification in the Universal Composability (UC) framework (see Section 2.2). This functionality $\mathcal{F}_{\text{PUBA}}$ (shown in Fig. 3.3) captures all security requirements discussed in Section 3.2.5. We use the standard UC model [Can01; Can20], and assume that the simulator $\mathcal{S}$ is activated by $\mathcal{F}_{\text{PUBA}}$ whenever any party provides any input.

Our functionality $\mathcal{F}_{\text{PUBA}}$ is defined such that all inputs have the form (Task name, List of secret inputs). The task name uniquely determines the task to be executed.

The notifications $\mathcal{S}$ obtains after $\mathcal{F}_{\text{PUBA}}$ receives input from any party depend on the respective party providing the input: On inputs from $\mathcal{TSA}$, O, or P, $\mathcal{F}_{\text{PUBA}}$ activates $\mathcal{S}$ with input (Task name, *pid*), where *pid* is the party identifier of $\mathcal{TSA}$, O, or P, respectively.

For users, however, we ensure unlinkability in all tasks except for UReg (for which we assume an out-of-band verification of the user's identity). During UReg, the functionality explicitly leaks pid_U of the calling U. For all other tasks, we model unlinkability by having the functionality only revealing the *role* of a user after a call, and *not* the *pid*; that is, the simulator only gets notified with (Task name, User) after a user sent input.

That way, with the exception of UReg, the *pid* of any user is never revealed to anyone outside of $\mathcal{F}_{\text{PUBA}}$. As such, the user can interact anonymously.

Whenever the function description states that $\mathcal{F}_{\text{PUBA}}$ performs some check, it aborts execution of that task if the check fails.

The stateful functionality Our ideal functionality is *stateful*, meaning that after interaction with any party, it updates its state. The state consists of several lists, which we present in Fig. 3.3. First of all, $\mathcal{F}_{\text{PUBA}}$ stores lists $\mathsf{L}_{\text{users}}$ of registered users. This list contains the PIDs of all users who registered using UReg, and which hence have valid logbooks. For each registered user, it stores the user's logbook in Log_U, which ensures that the only way to change it is by using the provided tasks.

For Analytics, we allow each user to outsource only one computation at a time. To that end, each user logbook Log_U has a flag that indicates whether it is currently outsourced. Furthermore, for each proxy the functionality keeps a list $\mathsf{L}_{\text{os}}^{\mathsf{P}}$ of all logbooks that were outsourced to it (containing the data at the point when Outsource was called). The results of the outsourced computations are stored in L_{UI}, until they are fetched by a user using $\mathcal{F}_{PUBA}^{(\Delta)}$-*Update*.

To ensure that computed functions conform to privacy requirements, the functionality stores all certified function parameters for a given task `task` in a list $\mathsf{L}_{\text{fp}}^{\texttt{task}}$. The only way to update this list is after a positive response from the trusted signing authority $\mathcal{TSA}$, which ensures that the operator can only perform computations of functions that were previously verified.

Functionality $\mathcal{F}_{\text{PUBA}}$

This functionality facilitates user-centric privacy-preserving analytics. The function to be computed is specified by the global parameter Δ.

State:
- List of all registered users $\mathsf{L}_{\text{users}}$
- List of all corrupted users L_{corr}
- List of all approved function parameters per task $\mathsf{L}_{\text{fp}}^{\texttt{task}}$
- Logbooks Log_U for each user
- For each proxy P a list of outsourced logbooks $\mathsf{L}_{\text{os}}^{\text{P}}$
- List of all pending updates L_{UI}
- Variable *initialized*, initially set to 0

Init:
- Input O: $(\texttt{init})$
- Input $\mathcal{TSA}$: $(\texttt{init})$
- Behavior:
  1. Set *initialized* $\coloneqq 1$
- Output O: $(\texttt{init}, \texttt{ok})$
- Output $\mathcal{TSA}$: $(\texttt{init}, \texttt{ok})$

The functionality only responds to the following inputs if *initialized* $= 1$

SignFP:
- Input O: $(\texttt{signfp}, ssid, fp, \texttt{task}, \texttt{in}_O)$
- Input $\mathcal{TSA}$: $(\texttt{signfp}, ssid, \texttt{in}_{\mathcal{TSA}})$
- Behavior:
  1. Check $\Delta(\texttt{signfp}, fp, \texttt{task}, \texttt{in}_O, \texttt{in}_{\mathcal{TSA}}) \stackrel{?}{=} 1$
  2. Check that fp have not been registered before for task $\texttt{task}$
  3. Let ℓ be $\min_\ell : (\cdot, \ell) \notin \mathsf{L}_{\text{fp}}^{\texttt{task}}$
  4. Add (fp, ℓ) to $\mathsf{L}_{\text{fp}}^{\texttt{task}}$
  5. Leak $(\texttt{task}, \ell)$ to the adversary
- Output both: $(\texttt{signfp}, ssid, \texttt{ok})$

UReg:
- Input U: $(\texttt{ureg}, ssid, \texttt{in}_U)$
- Input O: $(\texttt{ureg}, ssid, fp, \texttt{in}_O)$
- Behavior:
  1. If pid_U is already in $\mathsf{L}_{\text{users}}$, ignore this call
  2. Add pid_U to $\mathsf{L}_{\text{users}}$
  3. Let ℓ be $\min_\ell : (fp, \ell) \in \mathsf{L}_{\text{fp}}^{\texttt{bookkeep}(k)}$
  4. Leak ℓ to the adversary
  5. $(\mathcal{UH}, \texttt{out}_U, \texttt{out}_O) \leftarrow \Delta(\texttt{ureg}, fp, \texttt{in}_U, \texttt{in}_O)$
  6. Store $(pid_U, \mathcal{UH}, false)$ as Log_U
- Output U: $(\texttt{ureg}, ssid, \mathcal{UH}, \texttt{out}_U)$
- Output O: $(\texttt{ureg}, ssid, \texttt{out}_O)$

Bookkeeping:
- Input O: $(\texttt{bookkeep}, ssid, k, fp, \texttt{in}_O)$
- Input U: $(\texttt{bookkeep}, ssid, k, \texttt{in}_U)$
- Behavior:
  1. Let ℓ be $\min_\ell : (fp, \ell) \in \mathsf{L}_{\text{fp}}^{\texttt{bookkeep}(k)}$
  2. Leak ℓ to the adversary
  3. Set $(pid_U, \mathcal{UH}, outsourced) \coloneqq \text{Log}_U$
  4. $result \leftarrow \Delta(\texttt{bookkeep}, fp, k, \mathcal{UH}, \texttt{in}_U, \texttt{in}_O)$ with $result \coloneqq (\alpha, \mathbf{s}, \mathbf{a}, \texttt{out}_U, \texttt{out}_O)$
  5. $\mathcal{UH}' \leftarrow \alpha(\mathcal{UH})$.
  6. $\mathcal{UH}'' \coloneqq (uh_0'', \ldots, uh_{m-1}'')$ with
$$uh_i'' \coloneqq \begin{cases} \mathbf{s}[i] & \text{for } \mathbf{s}[i] \neq \perp \\ \mathcal{UH}'[i] & \text{for } \mathbf{s}[i] = \perp \end{cases}$$
  7. $\mathcal{UH}^{\text{new}} \coloneqq \mathcal{UH}'' + \mathbf{a}$
  8. Set $\text{Log}_U \coloneqq (pid_U, \mathcal{UH}^{\text{new}}, outsourced)$
- Output U: $(ssid, \mathcal{UH}, \texttt{out}_U)$
- Output O: $(ssid, \texttt{out}_O)$

Outsource:
- Input U: $(\texttt{outsource}, ssid, k, \texttt{in}_U)$
- Input P and O: $(\texttt{outsource}, ssid, k)$
- Behavior:
  1. If U and O are corrupted, append $(ssid, k, \perp, \perp, \perp)$ to $\mathsf{L}_{\text{os}}^{\text{P}}$ and skip all following steps
  2. Retrieve $(pid_U, \mathcal{UH}, outsourced) \coloneqq \text{Log}_U$
  3. Check $outsourced \stackrel{?}{=} false$
  4. Set $\text{Log}_U \coloneqq (pid_U, \mathcal{UH}, true)$
  5. Add $(ssid, k, pid_U, \mathcal{UH}, \texttt{in}_U)$ to $\mathsf{L}_{\text{os}}^{\text{P}}$
- Output all: $(ssid, \texttt{ok})$

Analytics:
- Input O: $(\texttt{analyt}, ssid, k, fp, \texttt{in}_O)$
- Input P: $(\texttt{analyt}, ssid, k)$
- Behavior:
  1. Let ℓ be $\min_\ell : (fp, \ell) \in \mathsf{L}_{\text{fp}}^{\texttt{analyt}(k)}$
  2. Leak ℓ to the adversary
  3. Load and remove the first Z_k entries $\left\{(ssid_z, k, pid_z, \mathcal{UH}_z, \texttt{in}_z)\right\}_{z=1}^{Z_k}$ from $\mathsf{L}_{\text{os}}^{\text{P}}$
  4. If P is corrupted, ask for updated inputs $\left\{\texttt{in}_z \mid 1 \le z \le Z_k, pid_z \in \mathsf{L}_{\text{corr}}\right\}$
  5. If O is corrupted, ask for updated information $(\mathcal{UH}_z, \texttt{in}_z)$ for each $z \in \{1, \ldots, Z_k\}$ with $pid_z \notin \mathsf{L}_{\text{users}}$
  6. $\texttt{out} \leftarrow \Delta(\texttt{analyt}, fp, k, \{(\mathcal{UH}_z, \texttt{in}_z)\}_{z=1}^{Z_k}, \texttt{in}_O)$
  7. $\left(\{(\alpha_z, \mathbf{s}_z, \mathbf{a}_z, \texttt{out}_z)\}_{z=1}^{Z_k}, \texttt{out}_O\right) \coloneqq \texttt{out}$
  8. Leak $\left\{(z, \alpha_z, \mathbf{s}_z, \mathbf{a}_z, \texttt{out}_z) \,\middle|\, 1 \le z \le Z_k, pid_z \in \mathsf{L}_{\text{corr}}\right\}$ to the adversary
  9. If O is corrupted, leak $\{(z, \alpha_z, \mathbf{s}_z, \mathbf{a}_z, \texttt{out}_z) \mid 1 \le z \le Z_k, pid_z \notin \mathsf{L}_{\text{users}}\}$ to the adversary
  10. For every z from 1 to Z_k with $pid_z \in \mathsf{L}_{\text{users}}$ add $(pid_z, ssid_z, \alpha_z, \mathbf{s}_z, \mathbf{a}_z, \texttt{out}_z)$ to L_{UI}
- Output P: $(ssid, \texttt{ok})$
- Output O: $(ssid, \{\alpha_z, \mathbf{s}_z\}_{z=1}^{Z_k}, \texttt{out}_O)$

Update:
- Input U, O, P: $(\texttt{update}, ssid)$
- Behavior:
  1. If U and O are corrupted, set $(\mathcal{UH}, \texttt{out}_U) = (\perp, \perp)$ and skip all following steps
  2. Load $(pid_U, \mathcal{UH}, outsourced) \coloneqq \text{Log}_U$
  3. Check $outsourced \stackrel{?}{=} true$
  4. Load and remove $(pid_U, ssid_{\text{OS}}, \alpha, \mathbf{s}, \mathbf{a}, \texttt{out}_U)$ from L_{UI}
  5. If O is corrupted, leak $ssid_{\text{OS}}$ to the adversary
  6. If U is honest and P is corrupted, leak $ssid_{\text{OS}}$ to the adversary
  7. $\mathcal{UH}' \leftarrow \alpha(\mathcal{UH})$
  8. $\mathcal{UH}'' \coloneqq (uh_0'', \ldots, uh_{m-1}'')$ with
$$uh_i'' \coloneqq \begin{cases} \mathbf{s}[i] & \text{for } \mathbf{s}[i] \neq \perp \\ \mathcal{UH}'[i] & \text{for } \mathbf{s}[i] = \perp \end{cases}$$
  9. Set $\text{Log}_U \coloneqq (pid_U, (\mathcal{UH}'' + \mathbf{a}), false)$
- Output U: $(\texttt{update}, ssid, (\alpha, \mathbf{s}, \mathbf{a}, \texttt{out}_U))$
- Output O: $(\texttt{update}, ssid, (\alpha, \mathbf{s}))$
- Output P: $(\texttt{update}, ssid, \texttt{ok})$

Figure 3.3.: The basic functionality $\mathcal{F}_{\text{PUBA}}$ for user-centric privacy-preserving analytics.

Tasks that are related to analytical computations, either directly (Bookkeeping) or outsourced (Outsource and Analytics) might be parameterized by multiple public function "frameworks" (the structure of the analytical function to execute, i.e., a logistic regression, a neural network with some set number of layers and neurons, a universal circuit of some size, etc.). We assume these are numerated and the tasks are given the identifier k of the concrete function to use. Additionally, when function parameters are certified, they are bound to one specific function (again identified by k) and only valid in that context. The identifier is relevant for Outsource to make sure the proxy only uses the outsourced user history for that specific function.

Init The initializing task from Fig. 3.3 must be invoked before anything else. This task serves to model required setup in the real protocol, such as the creation of cryptographic keys. The task only contains the operator and the TSA and essentially starts the whole process. Before invoking Init, all other calls are ignored; afterwards, the functionality responds to the other calls.

SignFP With this task, the operator can input function parameters fp to be used for a given task `task`. We assume that the application-specific function Δ contains some mechanism to verify that a given input fp is suitable for this task. If the function verifies fp for usage in `task`, then $\mathcal{F}_{\text{PUBA}}$ adds these fp to $\mathsf{L}_{\text{fp}}^{\texttt{task}}$, which enables its future use. The functionality leaks the new number of FPs it has for the task `task` to the adversary; we require this information for our simulation.

UReg With the UReg task depicted in Fig. 3.3, a user can register for participation. This task must be executed before any further interaction with the user. This is ensured in the ideal world by giving the user their initial user history and storing it in Log_U. For any future interaction, the user's current user history is always fetched from Log_U. This not only ensures that a user cannot manipulate data on their own, but it also enforces that any user wanting to perform a computation has undergone the registration process.

Specifics of the function that computes the initial user history are input as fp; a unique identifier—which does not leak any of the values in fp—is leaked to the adversary.

Bookkeeping The update of the latest user history (possibly based on the data collected so far) by a validly registered user is shown in Fig. 3.3. The functionality fetches the current user history from Log_U, which ensures that the latest user history is used. The operator inputs FPs fp, which define the function to be computed; the identifier of which is leaked to the adversary. We generally assume that computations leak only the basic structure, such as a neural network or logistic regression, to the user and hide the trade secrets in fp. To ensure that only valid fp are used, the functionality continues only if $fp \in \mathsf{L}_{\text{fp}}^{\texttt{bookkeep}(k)}$, and aborts if the operator attempts to use uncertified input.

The computation of function k is defined in the application-specific function Δ. It yields a result $(\alpha, \mathbf{s}, \mathbf{a}, \mathtt{out}_U, \mathtt{out}_O)$. $(\alpha, \mathbf{s}, \mathbf{a})$ is the part of the result used for the subsequent update of the user history. $\mathtt{out}_U$ and $\mathtt{out}_O$ are respective analytical outputs for the user and the operator. For our security proof we further require that the outputs of corrupted users are leaked to the adversary. However, we stress that this is not in conflict with our confidentiality guarantees, as we do not enforce privacy for corrupted users.

Afterwards, the user history is updated through "permute", "set", and "add" operations as described in Section 3.2.3. This 3-step process of updating the user history enables flexible updates, yet defines some rules for the update process so that later in the protocol (see Section 3.4.8) the user can efficiently prove that they correctly updated the user history.

Outsource The Outsource-task is described in Fig. 3.3. It is a three-party task, consisting of the user U providing the data, the operator O providing the framework, and a proxy P providing computational power.

In it, the functionality verifies that the user has no previous outsourced analytics computation in progress, and remembers that one was started now. It stores the current user history alongside the remaining information for later use, and marks it with the pid of the proxy that should be used for the computation. To allow linkability with the next Analytics and Update tasks, the functionality also stores the ssid of this task.

Analytics The actual analytical computation between operator and proxy is shown in Fig. 3.3. This requires that the required number of user logbooks have previously been outsourced through calls to Outsource. The functionality ensures this by fetching the values from its state, namely by taking the inputs of the users from $\mathsf{L}_{\text{os}}^{\mathsf{P}}$.

Similar to a normal computation, the proxy learns only a very basic structure, whereas the trade secrets involved in creating the function are hidden as input *fp* from O and only used if they were previously certified by $\mathcal{TSA}$ and hence stored in $\mathsf{L}_{\text{fp}}^{\mathtt{task}}$.

In addition to the computation of the function identified through k according to the application-specific function Δ and the subsequent storing of the results in L_{UI} for later use in $\mathcal{F}_{PUBA}^{(\Delta)}$-*Update*, the functionality contains leaks in case of corruptions. If both user and proxy are corrupted, we allow the adversary to input different auxiliary input $\mathtt{in}_U$. Furthermore, we stress that a corrupted operator can create arbitrary users, which means that for real protocols, it might be the case that some users who called $\mathcal{F}_{PUBA}^{(\Delta)}$-*Outsource* never registered. For those, we also ask the adversary for their inputs now, and additionally reveal their output directly after the computation.

Update The premise of our framework is to 1. ensure that any update to the user history is applied by the user, and 2. reward users who gave their data for analytics tasks. While relevant information was output during $\mathcal{F}_{PUBA}^{(\Delta)}$-*Analytics*, the final task of our functionality allows the user to fetch the results from $\mathcal{F}_{PUBA}^{(\Delta)}$-*Analytics* and obtain the incentive. Fig. 3.3

shows this step. The functionality fetches the latest user history (to model the fact that after Outsource, the user can perform further Bookkeeping tasks) and applies the three-stage update that was defined by the output from Analytics onto it. Furthermore, it marks the user as being allowed to outsource a computation again. If however both the user and operator are corrupted, we do not provide any guarantees and thus skip this step. Otherwise, we only leak the ssid of the corresponding outsource task to the adversary if at least one of operator or proxy is corrupted.

3.4. Instantiation

We now present the concrete instantiation Π_{PUBA} of PUBA. We begin by introducing some special notation (Section 3.4.1) and the cryptographic building blocks and the logbook data structure (Sections 3.4.2 and 3.4.3). We then explain the general principles underlying the protocol design (Section 3.4.4), and provide an overview of how the building blocks are combined to realize the informal security properties from Section 3.2.5 (Section 3.4.5). Then we provide an overview of the tasks and their workflow (Section 3.4.6), and describe the subfunctionality $\mathcal{F}_{\text{PPA}}$ that handles the computation of Δ (Section 3.4.7). The complete protocol specification is given in Section 3.4.8, followed by the formal security theorem (Section 3.4.9). The full proof of security is deferred to Section A.3.

3.4.1. Protocol-specific Notation

In addition to the general notation introduced in Chapter 2, we use the following protocol-specific notation throughout this section:

- com_{val} denotes a commitment to the value val
- $\overline{com}$ denotes a rerandomization of com
- o_{val} denotes a one-time pad suitable to encrypt the value val, and ct_{val} denotes the result of this encryption

Whenever the protocol states that a party performs some check, that party aborts the interaction if the check fails.

3.4.2. Cryptographic Building Blocks

This section gives an intuitive overview of the building blocks used in our construction. For formal definitions see Chapter 2.

Secure Channels Communication between the operator and the trusted signing authority, as well as with proxies, is done through secure channels. We use the UC-functionality $\mathcal{F}_{\text{SMT}}$ to model secure channels. A secure channel guarantees authenticity and confidentiality, and identifies both parties that partake in the communication. Communication between the operator and users only uses $\mathcal{F}_{\text{SMT}}$ during registration. For all other tasks, users should remain anonymous, and thus we make use of a different UC-functionality, $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$, which allows party A to establish a communication channel to party B, identifying party B but not revealing the identity of party A. It also guarantees authenticity and confidentiality of messages. For details and formal definitions, see Section 2.3 and Fig. 2.2.

A Structure-Preserving EUF-CMA-Secure Signature Scheme To ensure the integrity of the logbook, we use an EUF-CMA-secure signature scheme that is compatible (structure-preserving) with our zero-knowledge proofs. A possible instantiation is [Abe+11].

An Additively Homomorphic, Structure-Preserving Commitment Scheme Each user stores data inside their user history alongside additional information (see Section 3.4.3) to which they need to commit. We require the scheme to be homomorphic, unconditionally hiding and computationally binding. We require the additive homomorphism for two reasons: 1. Additively homomorphic commitments are efficiently rerandomizable. We use rerandomizing commitments to ensure that the commitments seen (and signed) by the operator after some interaction with the user are not the same as the ones used for the next interaction with that same user. Before any computation, the user has to prove authenticity of the user history. This requires the operator to at least learn commitments on the user history. After performing the computation the updated commitments need to be signed. If the user would send the old commitments directly to the next interaction, the operator could link the two tasks using the commitments. As a countermeasure we make heavy use of the rerandomization-trick: Instead of sending the commitments directly, parties reveal *rerandomized* commitments for the next interaction. We explain in Section 3.4.5 how this does not enable new attacks on the unforgeability of user histories. 2. For hiding the addition vector $\mathbf{a}$ from the operator we only output a *commitment* on this vector to the operator. The operator can then compute a commitment on the new user history homomorphically using only this output and the commitment on the old user history obtained by the user to compute the new signature. The user stores data in a logbook Log (more on that in Section 3.4.3), which contains data that is updated in every task. Letting the operator compute the commitment on the new user history homomorphically saves some rounds of communication and overall complexity. These requirements are satisfied by the commitments from [Abe+15].

A Non-Interactive Zero-Knowledge Proof-of-Knowledge Scheme This scheme is used by the user to prove that certain operations have been performed correctly. It needs to be extractable and zero-knowledge. Groth-Sahai proofs [GS08] satisfy our requirements.

A Robust Secret Sharing Scheme We use a robust secret sharing scheme, which lets a party create shares of a secret in such a way that 1. the recipients can verify the integrity of the received shares, and 2. tampering with the shares can be detected during reconstruction. Unlike *verifiable* secret sharing which only protects against a malicious dealer, *robust* secret sharing also protects against recipients trying to manipulate their shares in order to change the reconstructed output.

A Pairing Group All our schemes are defined over an asymmetric pairing group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, p, G_1, G_2)$. The groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order p are cyclic groups with generators G_1, G_2 and $e(G_1, G_2)$ respectively. Identification of a user relies on the hardness of the Co-CDH problem [BLS04], which asks to compute G_2^x given G_1, G_2, G_1^x. The Co-DH assumption is implied by the SXDH assumption [GS08] we use to instantiate Groth-Sahai proofs.

3.4.3. The User Logbook

We denote by Log the logbook containing the data stored by a user:

$$\mathsf{Log} = \left(\begin{matrix} \mathcal{UH} \\ com_{\mathcal{UH}} \\ decom_{\mathcal{UH}} \end{matrix}, \begin{matrix} ser \\ com_{ser} \\ decom_{ser} \end{matrix}, \begin{matrix} lin \\ com_{lin} \\ decom_{lin} \end{matrix}, \begin{matrix} id \\ com_{id} \\ decom_{id} \end{matrix}, \sigma \right) \tag{3.1}$$

The logbook contains all the data required to maintain the user history and to interact anonymously with the operator.

The User History $\mathcal{UH}$ The key component of the logbook is the user history. It is a vector $\mathcal{UH} \coloneqq (uh_0, \ldots, uh_{m-1})$ of m $\mathbb{Z}_p$ elements that represents the authenticated data collected by the user.

The Serial Number *ser* The serial number is a single $\mathbb{Z}_p$ element that uniquely determines a revision of a logbook. It is unique with overwhelming probability, in that throughout the lifetime of PUBA, there are no two logbooks (neither belonging to different users nor to the same user) that share the same serial number.

The Linking Number *lin* We require linkability *inside* the triplet for outsourcing computations: consecutive tasks of Outsource, Analytics and Update have to be linked to the same user as otherwise the data outsourced during the Outsource task cannot be used during Analytics and the changes of the resulting update cannot be applied to the correct user history during the Update task. We thus use an additional $\mathbb{Z}_p$ element that links these executions for all three parties involved in outsourcing computations, which the user stores inside the logbook. If no data has been outsourced since the latest update, the linking number is 0.

The Identity *id* The user has a fixed private identity which is represented as a secret $\mathbb{Z}_p$ element. It is randomly chosen during UReg and is never directly revealed to anybody. To prove ownership of the logbook, knowledge of *id* has to be proven.

The Commitment Information *com*, *decom* **and Signature** σ To ensure authenticity the operator generally signs all values inside the logbook. However, the signature is not on the values directly—as this would conflict with the user's privacy requirement—but on *commitments* thereof. This is why the user not only stores the values inside the logbook but also the commitments that were used by the operator to compute the signature. As those are part of the witness to generate zero-knowledge proofs, the user also stores the corresponding unveil information.

The final value in the logbook is a signature by the operator that ensures integrity of the commitments on the user history, the serial number, the linking number and the identity.

3.4.4. General Principles

For every task involving the user, our protocol begins and ends with the same two mechanisms: The authenticated input mechanism ensures the user enters a fresh and authenticated logbook into the interaction while the updating mechanism provides the user with a new valid logbook when the task is finished.

Authenticated Input Mechanism At the start of each task the user owns a valid logbook Log containing the data shown in Eq. (3.1). To prove the validity of the logbook to the operator, the user first *rerandomizes* all commitments com_* to commitments $\overline{com_*}$ by homomorphically adding a commitment to 0. In the second step, the user computes a zero-knowledge proof Π showing that they know 1. commitments com_* on the same values as the $\overline{com_*}$ and 2. a signature σ that authenticates the original commitments under the verification key of the operator. The rerandomized commitments $\overline{com_*}$ and proof Π are sent to the operator for validation. The above process is only conducted for values the operator is not supposed to learn, usually ($\mathcal{UH}$, *lin*, *id*). In case the user wants to fetch updates from an outsourced analytical computation or wants to start one and needs to show the logbook contains $lin = 0$, the hidden values are ($\mathcal{UH}$, *id*) only. The serial number *ser* is always revealed at the start of a task and checked by the operator (using a database lookup) to ensure that the user does not try to use an outdated logbook for a new task.

Updating Mechanism At the end of each task the user and operator jointly compute a new valid logbook Log^{new} to be used in the next task. To do so, the operator needs to reliably learn commitments ($com^{\text{new}}_{\mathcal{UH}}$, com^{new}_{ser}, com^{new}_{lin}, com^{new}_{id}) to all new values ($\mathcal{UH}^{\text{new}}$, lin^{new}, ser^{new}, *id*) and sign them for the user without learning the values themselves. We now explain how commitments to each of the values are obtained by the operator.

The new user history is calculated in two steps: Permutations and value setting by the user and additive increments by the operator. If there are updates which are not compatible with the homomorphic property of the underlying commitment scheme—i.e., permutations and data fields which should be set to entirely new values—the user performs those updates on their old user history, providing the operator with a new commitment value $com'_{\mathcal{UH}}$ to the updated version and a zero-knowledge proof that they have done so correctly. For updates compatible with the commitment scheme—i.e., additive increments—the operator directly learns a commitment $com_{\mathbf{a}}$ of the additive increment which they can add to $com'_{\mathcal{UH}}$ to obtain $com^{\text{new}}_{\mathcal{UH}}$. In case there are only additive updates, the first step is skipped and the operator directly adds $com_{\mathbf{a}}$ to the commitment $\overline{com_{\mathcal{UH}}}$ they learned at the start of the task.

A new serial number ser^{new} is coin-tossed similar to [Blu81] by the operator and user. The modified protocol works in two rounds: 1. The user picks a random share $ser^{\text{new}}_{(U)}$ and sends a commitment $com^{(U)}_{ser}$ of that value to the operator. 2. The operator picks a random share $ser^{\text{new}}_{(O)}$, computes a commitment $com^{(O)}_{ser}$ to that value, calculates the commitment $com^{\text{new}}_{ser} := com^{(U)}_{ser} + com^{(O)}_{ser}$ to the complete value $ser^{\text{new}} = ser^{\text{new}}_{(U)} + ser^{\text{new}}_{(O)}$ (without knowing $ser^{\text{new}}_{(U)}$ and without learning ser^{new} in the clear) and sends their serial number share alongside the commitment and unveil information to the user.

The linking number lin of the logbook stays the same for most interactions. Whenever the linking number is changed (in Outsource and Update) the operator knows the new value lin^{new}, so computing a new commitment com^{new}_{lin} to sign for the user is no problem. If $lin^{\text{new}} = lin$, the operator can use the rerandomized commitment $com^{\text{new}}_{lin} := \overline{com_{lin}}$ from the old logbook.

The ID id of course stays the same for one user throughout all tasks and hence the operator can use the rerandomized commitment $com^{\text{new}}_{id} := \overline{com_{id}}$ as well.

The operator signs all these commitments ($com^{\text{new}}_{\mathcal{UH}}$, com^{new}_{ser}, com^{new}_{lin}, com^{new}_{id}) to create a signature σ^{new} for the new logbook and sends it to the user. Finally, the user verifies the validity of the new logbook $\mathsf{Log}^{\text{new}}$.

3.4.5. Realization of Security Properties

Having introduced the building blocks, data structures, and general principles, we now explain at a high level how the protocol achieves the security properties described in Section 3.2.5. This provides a bridge between the informal security requirements and the formal protocol specification in Section 3.4.8.

Owner Binding Owner binding is achieved through the identity id stored in each logbook. The user proves knowledge of id in zero-knowledge during the authenticated input mechanism at the start of each task. Since id is randomly chosen during user registration and never revealed, only the legitimate owner can successfully prove knowledge of

this value. The operator's signature on the commitment to *id* ensures that this identity cannot be changed without detection.

History Freshness History freshness relies on the serial number *ser* mechanism. At the start of each task, the user reveals *ser* to the operator, who checks it against a database to ensure it has not been used before. At the end of each task, a new serial number ser^{new} is generated via a modified Blum coin-tossing protocol [Blu81] between user and operator, ensuring both parties contribute randomness. This guarantees that serial numbers are unique with overwhelming probability and that users cannot reuse old logbook states.

History Unforgeability Unforgeability is achieved through the operator's signature on commitments to all logbook values. The EUF-CMA security of the signature scheme ensures that users cannot create valid logbooks without the operator's participation. The authenticated input mechanism forces users to prove possession of a validly signed logbook at the start of each task.

The rerandomization of commitments (explained in Section 3.4.2) does not compromise unforgeability: while users can create fresh commitments to the same values, they must prove in zero-knowledge that these commitments contain the same values as the originally signed commitments. The computational binding property of the commitment scheme ensures that users cannot open rerandomized commitments to different values than those originally authenticated.

Uniqueness Uniqueness follows from the combination of history freshness and unforgeability. Since serial numbers are unique and checked by the operator, and since users cannot forge valid signatures on commitments, each user can maintain at most one valid logbook at any point in time.

Function Privacy Function privacy in outsourced computations is guaranteed through $\mathcal{F}_{\text{PPA}}$ and the hiding property of the commitments used: The sensitive function parameters are private inputs to $\mathcal{F}_{\text{PPA}}$ of the operator, and the surrounding protocol only handles commitments to them that were signed by the trusted signing authority $\mathcal{TSA}$. Details of this mechanism are provided in Section 3.4.6.

Unlinkability Unlinkability is achieved through commitment rerandomization and zero-knowledge proofs. At the start of each task, the user sends rerandomized commitments rather than the original signed commitments. Due to the unconditional hiding property of the commitment scheme, these rerandomized commitments are distributed independently of the original values and cannot be linked across tasks.

An exception is the triplet of Outsource-Analytics-Update tasks, which must be linkable to ensure computational results are applied to the correct user history. For this purpose, the logbook contains a linking number *lin* that is revealed during these three consecutive tasks but reset to 0 afterwards, ensuring unlinkability across different triplets.

Input Privacy Input privacy is achieved through the use of commitments and zero-knowledge proofs. The user never reveals the actual user history $\mathcal{UH}$ to the operator. Instead, they provide commitments to $\mathcal{UH}$ and prove properties about it in zero-knowledge.

The unconditional hiding property of the commitment scheme ensures that the operator learns nothing about the committed values beyond what is explicitly proven.

For additive updates to the user history, the operator learns only a commitment to the update vector $\mathbf{a}$ and can compute the updated commitment homomorphically without learning the actual values. For non-additive updates (permutations, value setting), the user performs the update locally and provides a zero-knowledge proof of correct execution.

Function Parameter Binding Function parameter binding in outsourced computations is ensured through signatures by a trusted signing authority ($\mathcal{TSA}$). The $\mathcal{TSA}$ signs commitments to function parameters, binding them to specific functions. The user/proxy verifies these signatures before performing computations, ensuring that only authorized function-parameter combinations are executed. The details of this mechanism are described in Sections 3.4.6 and 3.4.8.

3.4.6. Individual Tasks

Let us explain how the central tasks of Bookkeeping, Outsource, Analytics and Update are realized by Π_{PUBA}.

Bookkeeping The Bookkeeping task is almost completely covered by the authenticated input and updating mechanisms explained above. Between the two mechanisms, the user verifies that the operator uses correctly signed function parameters (cf. Section 3.2.2) for the task, and both of them jointly compute the update information and additional outputs according to Δ. We give more information on this MPC computation in Section 3.4.7.

Outsourcing Analytical Computations An overview of how Π_{PUBA} realizes outsourcing analytical computations can be found in Fig. 3.4.

For the Outsource task, the user reveals the logbook's linking number *lin* to be zero when proving the validity of their input logbook. Additionally, the user provides additive robust secret shares of their user history $\mathcal{UH}$ and their auxiliary input $\mathtt{in}_U$ to the analytics computation, and of several one-time pads for the proxy and operator, respectively. The one-time pads will be used to let the user collect their update information and computation outputs in a privacy-preserving manner. The secret shares are also proven to be computed correctly. Proxy and operator check the values using the robust secret sharing scheme and, using the values obtained from the proxy in this process, the operator checks the zero-knowledge proof to ensure that the shares were created correctly. Proxy and operator then coin toss a new linking number lin^{new} to be used for this Outsource/Analytics/Update triple. Again, as the last step the operator provides the user with the necessary information to obtain a new valid logbook, which only contains a new serial and linking number but the same user history as before.

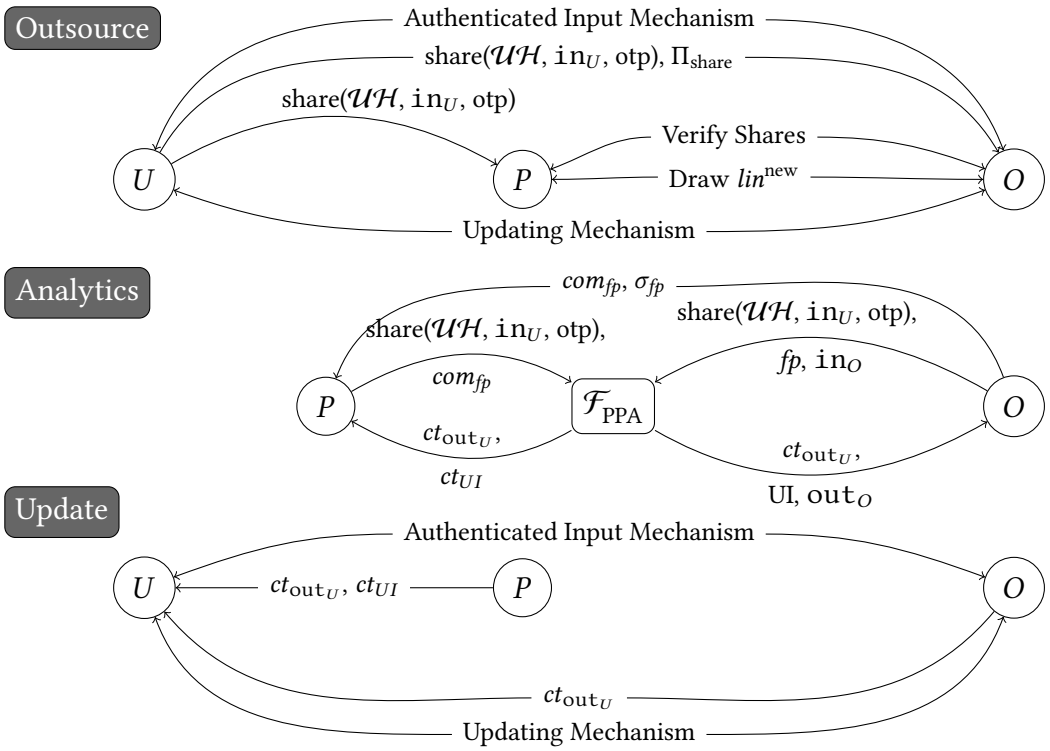

Figure 3.4.: Simplified depiction of Π_{PUBA} for outsourcing computations.

As Analytics does not include the user, the general principles from Section 3.4.4 do not apply to this task. Instead, the proxy verifies that the operator uses correctly signed FPs fp for the computation. To do so, the operator sends a commitment com_{fp} as well as a signature σ_{fp} on the commitment to the proxy, who checks that the signature verifies under the key of the TSA. Afterwards both jointly compute Δ via $\mathcal{F}_{\text{PPA}}$. Note that—apart from the update information the operator is supposed to learn—the update information and outputs for the respective users are one-time padded to retain privacy from both proxy and operator.

In the Update task, the user again proves their input logbook to be valid, revealing the linking number lin so the proxy and operator can hand out the correct update information and user output from Analytics. The user unmasks the one-time padded information from the proxy and operator, respectively, checking them for consistency. The remainder of the task again consists of the user and operator conducting the logbook updating mechanism. The linking number lin^{new} is set to zero again in this process so the user will be able to outsource a new analytical task.

3.4.7. Wrapping the Computation of Δ

Our protocol requires certain pre- and post-processing steps, before the actual MPC-protocol can be executed. We model those additional steps, namely how the inputs and

Functionality $\mathcal{F}_{\text{PPA}}$

This functionality performs the computation specified by Δ in a way that provides *input consistency*. It is parameterized by the to-be-computed function Δ, a *homomorphic commitment scheme* COM which is *unconditionally hiding* and *computationally binding* and uses the subprotocol Π_{Share} with the same commitment scheme COM.

UReg:

- Input U: $(\texttt{ureg}, ssid, com_{fp}, \texttt{in}_U)$
- Input O: $(\texttt{ureg}, ssid, fp, decom_{fp}, \texttt{in}_O)$
- Behavior:
  1. Check open$(com_{fp}, decom_{fp}, fp) \stackrel{?}{=} 1$
  2. Compute $result \leftarrow \Delta(\texttt{ureg}, fp, \texttt{in}_U, \texttt{in}_O)$ with $result \coloneqq (\mathcal{UH}, \texttt{out}_U, \texttt{out}_O)$
  3. $(com_{\mathcal{UH}}, decom_{\mathcal{UH}}) \leftarrow \text{commit}(\mathcal{UH})$
- Output U: $(ssid, \mathcal{UH}, decom_{\mathcal{UH}}, \texttt{out}_U)$
- Output O: $(ssid, com_{\mathcal{UH}}, \texttt{out}_O)$

Bookkeeping:

- Input U: $(\texttt{bookkeep}, ssid, k, \mathcal{UH}, decom_{\mathcal{UH}}, com_{fp}, \texttt{in}_U)$
- Input O: $(\texttt{bookkeep}, ssid, k, com_{\mathcal{UH}}, fp, decom_{fp}, \texttt{in}_O)$
- Behavior:
  1. Check open$(com_{\mathcal{UH}}, decom_{\mathcal{UH}}, \mathcal{UH}) \stackrel{?}{=} 1$.
  2. Check open$(com_{fp}, decom_{fp}, fp) \stackrel{?}{=} 1$.
  3. Compute $result \leftarrow \Delta(\texttt{bookkeep}, fp, k, \mathcal{UH}, \texttt{in}_U, \texttt{in}_O)$ with $result \coloneqq (\alpha, \mathbf{s}, \mathbf{a}, \texttt{out}_U, \texttt{out}_O)$
  4. $(com_{\mathbf{a}}, decom_{\mathbf{a}}) \leftarrow \text{commit}(\mathbf{a})$.
- Output U: $(ssid, \alpha, \mathbf{s}, \mathbf{a}, com_{\mathbf{a}}, decom_{\mathbf{a}}, \texttt{out}_U)$
- Output O: $(ssid, \alpha, \mathbf{s}, com_{\mathbf{a}}, \texttt{out}_O)$

Analytics:

- Input P: $(\texttt{analyt}, ssid, k, com_{fp}, \{\langle \text{Log} \rangle^P\}_{z=1}^{Z_k})$
- Input O: $(\texttt{analyt}, ssid, k, fp, decom_{fp}, \{\langle \text{Log} \rangle^O\}_{z=1}^{Z_k}, \texttt{in}_O)$
- Behavior:
  1. Check open$(com_{fp}, decom_{fp}, fp) \stackrel{?}{=} 1$
  2. For every z from 1 to Z_k:
     - $\mathcal{UH}_z \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \mathcal{UH}_z \rangle^P, \langle \mathcal{UH}_z \rangle^O)$
     - $\texttt{in}_{U_z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \texttt{in}_{U_z} \rangle^P, \langle \texttt{in}_{U_z} \rangle^O)$
     - $\texttt{o}_{\text{OUT}\,z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \texttt{o}_{\text{OUT}\,z} \rangle^P, \langle \texttt{o}_{\text{OUT}\,z} \rangle^O)$
     - $\texttt{o}_{\text{UNV}\,z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \texttt{o}_{\text{UNV}\,z} \rangle^P, \langle \texttt{o}_{\text{UNV}\,z} \rangle^O)$
     - $\texttt{o}_{\alpha z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \texttt{o}_{\alpha z} \rangle^P, \langle \texttt{o}_{\alpha z} \rangle^O)$
     - $\texttt{o}_{\mathbf{s}z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \texttt{o}_{\mathbf{s}z} \rangle^P, \langle \texttt{o}_{\mathbf{s}z} \rangle^O)$
     - $\texttt{o}_{\mathbf{a}z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \texttt{o}_{\mathbf{a}z} \rangle^P, \langle \texttt{o}_{\mathbf{a}z} \rangle^O)$
  3. Compute $result \leftarrow \Delta(\texttt{analyt}, fp, k, \{(\mathcal{UH}_z, \texttt{in}_{U_z})\}_{z=1}^{Z}, \texttt{in}_O)$ with $result \coloneqq (\{(\alpha_z, \mathbf{s}_z, \mathbf{a}_z, \texttt{out}_{U_z})\}_{z=1}^{Z_k}, \texttt{out}_O)$
  4. For every z from 1 to Z_k:
     - $(com_{\mathbf{a}_z}, decom_{\mathbf{a}_z}) \leftarrow \text{commit}(\mathbf{a}_z)$
     - $ct_{decom_{\mathbf{a}_z}} \coloneqq decom_{\mathbf{a}_z} + \texttt{o}_{\text{UNV}\,z}$
     - $ct_{\texttt{out}_{U_z}} \coloneqq \texttt{out}_{U_z} + \texttt{o}_{\text{OUT}\,z}$
     - $ct_{\alpha_z} \coloneqq \alpha_z + \texttt{o}_{\alpha z}$
     - $ct_{\mathbf{s}_z} \coloneqq \mathbf{s}_z + \texttt{o}_{\mathbf{s}z}$
     - $ct_{\mathbf{a}_z} \coloneqq \mathbf{a}_z + \texttt{o}_{\mathbf{a}z}$
- Output P: $(ssid, \{(ct_{\alpha_z}, ct_{\mathbf{s}_z}, ct_{\mathbf{a}_z}, ct_{decom_{\mathbf{a}_z}}, ct_{\texttt{out}_{U_z}})\}_{z=1}^{Z_k})$
- Output O: $(ssid, \{(\alpha_z, \mathbf{s}_z, com_{\mathbf{a}_z}, ct_{\texttt{out}_{U_z}})\}_{z=1}^{Z_k}, \texttt{out}_O)$

Figure 3.5.: The functionality $\mathcal{F}_{\text{PPA}}$ we use to perform the computations.

outputs are handled, by an additional *subfunctionality* $\mathcal{F}_{\text{PPA}}$. We constructed our protocol in a modular way. While our contribution lies in the construction of a privacy-preserving bookkeeping mechanism built *around* any existing MPC framework, we move the correct usage of the MPC framework itself into an own subfunctionality $\mathcal{F}_{\text{PPA}}$, depicted in Fig. 3.5.

The key reason why we cannot use a general MPC framework directly is that we require extra steps to ensure that both parties input the same data that was previously verified by our protocol. However, those extra steps can be achieved by using any protocol for secret-sharing-based MPC and expanding the computed function accordingly. Moreover, we stress that for a given instantiation of Δ, more efficient instantiations of $\mathcal{F}_{\text{PPA}}$ making non-black-box use of Δ (for example by letting the user prove in zero-knowledge that the function was computed correctly) are also possible.

We use the underlying computation mechanism to extend the following tasks:

UReg Here, user and operator use the actual MPC framework for computing the initial user history based on the (unauthenticated) inputs of both parties. Essentially, the added

code verifies, before the computation, that the operator used the same FPs for the MPC that were also verified earlier. This is done by having the operator input unveil information $decom_{fp}$ in addition to their function parameters fp, which opens a commitment com_{fp} input by the user. After the actual output of Δ has been computed, we require additional steps to compute the initial commitment on the user history; from $\mathcal{F}_{\text{PPA}}$, the operator only learns the commitment on the user history and not the actual values.

Bookkeeping Before starting the actual computation of Δ, $\mathcal{F}_{\text{PPA}}$ verifies that both sides used the correct values as input; verifying the function parameters fp used by the operator works the same as during UReg. Verification of the user inputs similarly requires the user to input the user history $\mathcal{UH}$ and unveil information $decom_{\mathcal{UH}}$, which opens a commitment $com_{\mathcal{UH}}$ input by O. $\mathcal{F}_{\text{PPA}}$ only continues with computing the actual function Δ on the given inputs if the commitments successfully opened to the specified values. Furthermore, the additive increment of the update should not be learned by the operator; thus, a commitment on this vector is generated. The operator then only learns the commitment and the user learns the plain value and unveil information.

Analytics The function parameters are verified the same way as during UReg and Bookkeeping. Our protocol for outsourcing the latest user history is shown in Fig. 3.4: During the Outsource task, it lets the user create robust secret shares of their user history for the proxy and the operator and *prove* that those are indeed shares which sum up to an authenticated user history. The proxy and operator then only insert their respective shares and the wrapper ensures that 1. the shares are reconstructed correctly, and 2. computation only continues if no tampering is detected. After successful verification, $\mathcal{F}_{\text{PPA}}$ internally reconstructs the shares for each user, computes the function, and *masks* the outputs. This means that $\mathcal{F}_{\text{PPA}}$ only outputs *cryptographically protected* private outputs $\texttt{out}_U$ for each participating user. $\mathcal{F}_{\text{PPA}}$ also provides the proxy P with additional information for the user to verify the integrity of the update. It makes use of the protocol Π_{Share} given in Fig. 3.6.

The function is designed to output the required maps for our three-stage update mechanism: the permutation α, the direct update $\mathbf{s}$, and the additive increment $\mathbf{a}$. Since the operator is again not supposed to learn $\mathbf{a}$, the wrapper adds code that computes an honest commitment and corresponding unveil information. As the user cannot fetch their relevant output directly, those are routed through the proxy. To ensure that the proxy does not learn any of the values in the process, we let the user not only create robust shares of the user history but also of five one-time pads (OTPs). The first three OTPs hide α, $\mathbf{s}$, and $\mathbf{a}$, respectively. The fourth one hides the decommitment information on $\mathbf{a}$, and the final one is used to mask the function outputs relevant for the user. The additional code hence has to reconstruct and apply these masks to the function output and output all these masked values to the proxy. This hides all information from the proxy P, as only the user knows the one-time pads.

Protocol Π_{Share}

This protocol facilitates robust secret sharing. It is parameterized by an *unconditionally hiding* and *computationally binding homomorphic commitment scheme* COM, for which inverse elements in the image group of COM are efficiently computable.

Share:

- Input: x
- Behavior:
  1. Sample uniformly random $x^{(0)}$
  2. $x^{(1)} \coloneqq x - x^{(0)}$
  3. $(com_x^{(0)},\ decom_x^{(0)}) \leftarrow \text{commit}(x^{(0)})$
  4. $(com_x^{(1)},\ decom_x^{(1)}) \leftarrow \text{commit}(x^{(1)})$
  5. $\langle x \rangle^0 \coloneqq \left(x^{(0)}, com_x^{(1)}, decom_x^{(0)}\right)$
  6. $\langle x \rangle^1 \coloneqq \left(x^{(1)}, com_x^{(0)}, decom_x^{(1)}\right)$
- Output: $(\langle x \rangle^0,\ \langle x \rangle^1)$

Combine:

- Input: $(\langle x \rangle^0,\ \langle x \rangle^1)$
- Behavior:
  1. Parse $(x^{(0)},\ com_x^{(1)},\ decom_x^{(0)}) \coloneqq \langle x \rangle^0$
  2. Parse $(x^{(1)},\ com_x^{(0)},\ decom_x^{(1)}) \coloneqq \langle x \rangle^1$
  3. Check $\text{open}(com_x^{(0)}, decom_x^{(0)}, x^{(0)}) \stackrel{?}{=} 1$
  4. Check $\text{open}(com_x^{(1)}, decom_x^{(1)}, x^{(1)}) \stackrel{?}{=} 1$
  5. $x \coloneqq x^{(0)} + x^{(1)}$
- Output: x

Figure 3.6.: Protocol for robust secret sharing.

3.4.8. Complete Protocol Specification

Additionally to the subfunctionality $\mathcal{F}_{\text{PPA}}$, we make use of two subprotocols that encapsulate recurring tasks in our protocol.

A robust secret-sharing-protocol Both our subfunctionality and our protocol make use of a *sharing* protocol and its corresponding *combine* protocol. Those are required so that the user can *share* information with P and O in such a way, that no party —neither P nor O —can change the shares unnoticed. The protocols are shown in Fig. 3.6.

Essentially, this comes down to *additive secret sharing*, where the dealer does not only send the additive share of a value to each party, but also adds a *commitment* on the respective *other party's* share. Additionally, each party obtains unveil information on its own commitment.

The combine protocol then lets each party send their own value and their unveil information; the other party only accepts if the unveil information matches the commitment it received during the sharing-phase.

Logbook verification Additionally, since this task is used after each user interaction during any task, we have a *verification* protocol Π_{Verify}, shown in Fig. 3.7. This protocol lets a user check that their logbook Log has been created correctly.

The protocol basically verifies that all the commitments are on the correct values and that the signature is correct.

Additionally, we also encapsulated the process of sharing the whole logbook (using Π_{Share}) and verifying those shares into Π_{Verify}.

Protocol Π_{Verify}

This protocol facilitates sharing and verification of logbooks. It is parameterized by an *unconditionally hiding* and *computationally binding homomorphic commitment scheme* COM, for which inverse elements in the image group of COM are efficiently computable, and an EUF-CMA-secure, structure-preserving signature scheme SIG.

Verify Logbook:

- Input: $(\mathtt{VfyL}, \mathsf{Log} \coloneqq (\mathcal{UH}, com_{\mathcal{UH}}, decom_{\mathcal{UH}}, ser, com_{ser}, decom_{ser}, lin, com_{lin}, decom_{lin}, id, com_{id}, decom_{id}, \sigma))$
- Behavior:
  1. Check $\mathsf{open}(com_{\mathcal{UH}}, decom_{\mathcal{UH}}, \mathcal{UH}) \stackrel{?}{=} 1$
  2. Check $\mathsf{open}(com_{ser}^{(O)}, decom_{ser}^{(O)}, ser^{(O)}) \stackrel{?}{=} 1$
  3. Check $\mathsf{open}(com_{lin}, decom_{lin}, lin) \stackrel{?}{=} 1$
  4. Check $\mathsf{open}(com_{id}, decom_{id}, id) \stackrel{?}{=} 1$
  5. Check $\mathsf{SIG.Vf}(\mathsf{vk}_O, (com_{\mathcal{UH}}, com_{ser}, com_{lin}, com_{id}), \sigma) \stackrel{?}{=} 1$
- Output: (Log)

Share Logbook:

- Input: $(\mathtt{ShareL}, \mathsf{Log} \coloneqq (\mathcal{UH}, com_{\mathcal{UH}}, decom_{\mathcal{UH}}, ser, com_{ser}, decom_{ser}, lin, com_{lin}, decom_{lin}, id, com_{id}, decom_{id}, \sigma))$
- Behavior:
  1. $\mathsf{o}_{\text{OUT}} \xleftarrow{r} \mathcal{IN}$, $\mathsf{o}_{\text{UNV}} \xleftarrow{r} \mathbb{Z}_p$
  2. $\mathsf{o}_\alpha \xleftarrow{r} \mathbb{Z}_p^m$, $\mathsf{o_s} \xleftarrow{r} \mathbb{Z}_p^m$, $\mathsf{o_a} \xleftarrow{r} \mathbb{Z}_p^m$
  3. $(\langle \mathcal{UH} \rangle^P, \langle \mathcal{UH} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathcal{UH})$
  4. $(\langle \mathtt{in}_U \rangle^P, \langle \mathtt{in}_U \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathtt{in}_U)$
  5. $(\langle \mathsf{o}_{\text{OUT}} \rangle^P, \langle \mathsf{o}_{\text{OUT}} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathsf{o}_{\text{OUT}})$
  6. $(\langle \mathsf{o}_{\text{UNV}} \rangle^P, \langle \mathsf{o}_{\text{UNV}} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathsf{o}_{\text{UNV}})$
  7. $(\langle \mathsf{o}_\alpha \rangle^P, \langle \mathsf{o}_\alpha \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathsf{o}_\alpha)$
  8. $(\langle \mathsf{o_s} \rangle^P, \langle \mathsf{o_s} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathsf{o_s})$
  9. $(\langle \mathsf{o_a} \rangle^P, \langle \mathsf{o_a} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathsf{o_a})$
  10. $\langle \mathsf{Log} \rangle^X \coloneqq (\langle \mathcal{UH} \rangle^X, \langle \mathtt{in}_U \rangle^X, \langle \mathsf{o}_{\text{OUT}} \rangle^X, \langle \mathsf{o}_{\text{UNV}} \rangle^X, \langle \mathsf{o}_\alpha \rangle^X, \langle \mathsf{o_s} \rangle^X, \langle \mathsf{o_a} \rangle^X)$ for $X \in \{P, O\}$
- Output: $(\mathsf{o}_{\text{OUT}}, \mathsf{o}_{\text{UNV}}, \mathsf{o}_\alpha, \mathsf{o_s}, \mathsf{o_a}, \langle \mathsf{Log} \rangle^O, \langle \mathsf{Log} \rangle^P)$

Verify Shares:

- Input: $(\mathtt{VfyS}, \langle \mathsf{Log} \rangle^X, VerInf(\mathsf{Log})^Y)$ with $X \in \{P, O\}$ and $Y \in \{P, O\} \setminus \{X\}$
- Behavior:
  1. $VerInf(\mathsf{Log})^Y =: (com_{\mathcal{UH}}^{(Y)}, com_{\mathtt{in}_U}^{(Y)}, com_{\mathsf{o}_{\text{OUT}}}^{(Y)}, com_{\mathsf{o}_{\text{UNV}}}^{(Y)}, com_{\mathsf{o}_\alpha}^{(Y)}, com_{\mathsf{o_s}}^{(Y)}, com_{\mathsf{o_a}}^{(Y)})$
  2. Check $\mathsf{open}(com_{\mathcal{UH}}^{(Y)}, decom_{\mathcal{UH}}^{(X)}, \mathcal{UH}^{(X)}) \stackrel{?}{=} 1$
  3. Check $\mathsf{open}(com_{\mathtt{in}_U}^{(Y)}, decom_{\mathtt{in}_U}^{(X)}, \mathtt{in}_U^{(X)}) \stackrel{?}{=} 1$
  4. Check $\mathsf{open}(com_{\mathsf{o}_{\text{OUT}}}^{(Y)}, decom_{\mathsf{o}_{\text{OUT}}}^{(X)}, \mathsf{o}_{\text{OUT}}{}^{(X)}) \stackrel{?}{=} 1$
  5. Check $\mathsf{open}(com_{\mathsf{o}_{\text{UNV}}}^{(Y)}, decom_{\mathsf{o}_{\text{UNV}}}^{(X)}, \mathsf{o}_{\text{UNV}}{}^{(X)}) \stackrel{?}{=} 1$
  6. Check $\mathsf{open}(com_{\mathsf{o}_\alpha}^{(Y)}, decom_{\mathsf{o}_\alpha}^{(X)}, \mathsf{o}_\alpha{}^{(X)}) \stackrel{?}{=} 1$
  7. Check $\mathsf{open}(com_{\mathsf{o_s}}^{(Y)}, decom_{\mathsf{o_s}}^{(X)}, \mathsf{o_s}{}^{(X)}) \stackrel{?}{=} 1$
  8. Check $\mathsf{open}(com_{\mathsf{o_a}}^{(Y)}, decom_{\mathsf{o_a}}^{(X)}, \mathsf{o_a}{}^{(X)}) \stackrel{?}{=} 1$
- Output: $(\mathtt{ok})$

Figure 3.7.: Protocol for user logbook verification.

3.4.8.1. The Protocol

With those building blocks we can now provide our full protocol. The protocol runs in the $\{\mathcal{F}_{\text{PPA}}, \mathcal{F}_{\text{BB}}, \mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{SMT}}^{\text{os-auth}}, \mathcal{F}_{\text{CRS}}\}$-hybrid model. Those are used for the following purpose:

$\mathcal{F}_{\text{PPA}}$ has been explained in Fig. 3.5.

$\mathcal{F}_{\text{BB}}$ is used during user registration to ensure that no user creates more than one account and to verify that the user really is who they claim to be and is given in Fig. 2.1b.

$\mathcal{F}_{\text{SMT}}$/$\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ are used to provide secure and authenticated communication between parties and are given in Figs. 2.2a and 2.2b.

$\mathcal{F}_{\text{CRS}}$ is required for our pairing-based instantiations of the zero-knowledge protocol and is given in Fig. 2.1a.

The full protocol is parameterized by:

- group parameters gp specifying pairing-friendly curves over which all other building blocks are instantiated.

Protocol $\Pi_{\text{PUBA}}(1^\lambda, n, t)$, Part 1

Party State:

- Every party stores:
  - Common reference strings (crs_{ZK}, crs_{com})
- The operator O stores:
  - Signature key pair $(\text{vk}_O, \text{sk}_O)$
  - List L_{users} of registered user public keys
  - List L_{SER} of observed serial numbers
  - Mapping $f_{\text{OI}}^{(O)}$ on $\{pid_P\} \times \{1, \ldots, K\}$ that maps (pid_P, k) to a list $f_{\text{OI}}^{(O)}(pid_P, k)$ of entries $(lin, \langle \mathcal{UH} \rangle^O, \langle \texttt{in}_U \rangle^O, \langle \text{o}_{\text{OUT}} \rangle^O, \langle \text{o}_\alpha \rangle^O, \langle \text{o}_\text{s} \rangle^O, \langle \text{o}_\text{a} \rangle^O)$
  - Partial mapping $f_{\text{UI}}^{(O)}$ on $\{lin\}$: $lin \mapsto (\alpha, \text{s}, com_\text{a}, ct_{\text{out}_U})$
  - List $\text{L}_{\text{fp}}^{\texttt{task}}$ of certified function parameters fp together with the tuple $(com_{fp}, decom_{fp}, \sigma_{fp})$
- Each proxy P stores:
  - Verification key vk_O of the operator
  - Verification key $\text{vk}_{\mathcal{TSA}}$ of the trusted signing authority
  - Mapping $f_{\text{OI}}^{(P)}$ on $\{1, \ldots, K\}$ that maps k to a list $f_{\text{OI}}^{(P)}(k)$ of entries $(lin, \langle \mathcal{UH} \rangle^P, \langle \texttt{in}_U \rangle^P, \langle \text{o}_{\text{OUT}} \rangle^P, \langle \text{o}_\alpha \rangle^P, \langle \text{o}_\text{s} \rangle^P, \langle \text{o}_\text{a} \rangle^P)$
  - Partial mapping $f_{\text{UI}}^{(P)}$ on $\{lin\}$: $lin \mapsto (ct_\alpha, ct_\text{s}, ct_\text{a}, ct_{\text{out}_U})$
- Each user U stores:
  - User logbook Log
  - User public key pk_U
  - One-time pads o_α, o_s, o_a, o_{OUT} and o_{UNV}
  - Verification key vk_O of the operator
  - Verification key $\text{vk}_{\mathcal{TSA}}$ of the trusted signing authority

Init:

- Input O: $(\texttt{init})$
- Input $\mathcal{TSA}$: $(\texttt{init})$
- Behavior:
  - O: $(\text{vk}_O, \text{sk}_O) \leftarrow \text{SIG.Keygen}(\text{gp})$
  - O: Send $(\texttt{Register}, vko)$ to $\mathcal{F}_{\text{BB}}$
  - $\mathcal{TSA}$: $(\text{vk}_{\mathcal{TSA}}, \text{sk}_{\mathcal{TSA}}) \leftarrow \text{SIG.Keygen}(\text{gp})$
  - $\mathcal{TSA}$: Send $(\texttt{Register}, \text{vk}_{\mathcal{TSA}})$ to $\mathcal{F}_{\text{BB}}$
- Output both: $(\texttt{init}, \text{ok})$

SignFP:

- Input O: $(\texttt{signfp}, ssid, fp, \texttt{task}, \texttt{in}_O)$
- Input $\mathcal{TSA}$: $(\texttt{signfp}, ssid, \texttt{in}_{\mathcal{TSA}})$
- Behavior:
  - O: $(com_{fp}, decom_{fp}) \leftarrow \text{commit}(fp)$
  - O: Send $(ssid \| 1, pid_{\mathcal{TSA}}, (fp, com_{fp}, decom_{fp}, \texttt{task}, \texttt{in}_O))$ to $\mathcal{F}_{\text{SMT}}$
  - $\mathcal{TSA}$: Receive $(ssid \| 1, pid_O, (fp, com_{fp}, decom_{fp}, \texttt{task}, \texttt{in}_O))$ from $\mathcal{F}_{\text{SMT}}$
  - $\mathcal{TSA}$: Check $\Delta(\texttt{signfp}, fp, \texttt{task}, \texttt{in}_O, \texttt{in}_{\mathcal{TSA}}) \stackrel{?}{=} 1$
  - $\mathcal{TSA}$: Check $\text{open}(com_{fp}, decom_{fp}, fp) \stackrel{?}{=} 1$
  - $\mathcal{TSA}$: $\sigma_{fp} \leftarrow \text{Sign}(\text{sk}_{\mathcal{TSA}}, (\texttt{task}, com_{fp}))$
  - $\mathcal{TSA}$: Send $(ssid \| 2, pid_O, (\sigma_{fp}))$ to $\mathcal{F}_{\text{SMT}}$
  - O: Receive $(ssid \| 2, pid_{\mathcal{TSA}}, (\sigma_{fp}))$ from $\mathcal{F}_{\text{SMT}}$
  - O: Add $(fp, com_{fp}, decom_{fp}, \sigma_{fp})$ to $\text{L}_{\text{fp}}^{\texttt{task}}$
- Output both: $(\texttt{signfp}, ssid, \text{ok})$

Figure 3.8.: Protocol Π_{PUBA} part 1, state and initialization.

- an *unconditionally hiding* and *computationally binding homomorphic commitment scheme* COM for which inverse elements in the image group of COM are efficiently computable. This is instantiated as the same scheme in Π_{Share} and Π_{Verify}.
- an EUF-CMA-secure, structure-preserving *signature scheme* SIG. Again, this scheme is the same as the one used for Π_{Verify}.
- a trapdoor dual-mode zero-knowledge proof-of-knowledge scheme POK based on a common reference string crs_{ZK} where an honestly chosen crs_{ZK} yields overwhelming completeness and negligible soundness error whereas one mode offers F-extractability and the other mode offers simulatability.

The protocol from Figs. 3.8, 3.9, 3.11 and 3.13 makes heavy use of the languages defined in Figs. 3.10, 3.12a, 3.12b, 3.14a and 3.14b. The intuition behind each task is as follows:

Init Before any other task can be executed, the operator and the TSA have to run the Init task shown in Fig. 3.8. While we modeled it as a two-party task, the two parties never interact with one another and only create their signing keys and register them at $\mathcal{F}_{\text{BB}}$.

SignFP Our framework is designed such that it hides the function details from the user. While we believe this to be extremely important on one hand, as the operator potentially spent a lot of time and money in the creation of the model, we stress that fully hiding the function *and* the result from the user allows for a trivial attack on the user's anonymity if the operator inputs a non-privacy-preserving function such as the identity.

As a compromise we suggest a third party which *certifies* that the operator's input meets certain privacy criteria: the trusted signing authority (TSA) $\mathcal{TSA}$. The key idea is that the to-be-computed function only has a very generic design—say in the form of general logistic regression or a neural network—but the function specifics are stored in function parameters *fp*. In order to circumvent the attack sketched above and ensure that the operator only ever uses *valid* inputs, the tasks UReg, Bookkeeping, and Analytics require *signed* FPs. To that end, the task SignFP shown in Fig. 3.8 lets the operator input some FPs *fp* which are to be used for computation of some task $\texttt{task} \in \{\texttt{ureg}, (\texttt{bookkeep}, k), (\texttt{analyt}, k)\}$ alongside a commitment com_{fp} and corresponding unveil information $decom_{fp}$. Here, k identifies which of the possibly multiple generic function designs for the given task these function parameters belong to. The TSA verifies them using the application-dependent Δ to ensure that they match the required privacy standards. We assume those privacy standards to be public knowledge.

If a given set of FPs verifies, the operator obtains a signature σ_{fp} on $(\texttt{task}, com_{fp})$ and uses com_{fp} as future certificate for *fp*: Before using the FPs *fp*, the O sends com_{fp} and σ_{fp} to the respective other party (the user or the proxy, respectively), who then inputs com_{fp} into $\mathcal{F}_{\text{PPA}}$. The operator O inputs *fp* and $decom_{fp}$. Before starting the computation defined by Δ, the commitment is verified. The binding property of the commitment scheme COM ensures that the operator can only use FPs which were signed by $\mathcal{TSA}$.

UReg The protocol for user registration is shown in Fig. 3.9. We want to ensure that each user has at most a single logbook. To ensure that no user can create multiple accounts we make use of the *Bulletin Board* functionality $\mathcal{F}_{\text{BB}}$. This lets the user register a key *exactly once*. The user has to publish a fresh *public key* to the bulletin board which is taken from the pairing group given in gp. The user can only register if they know the corresponding secret key and hence prove knowledge of it using the zero-knowledge proof from Fig. 3.10. This proves knowledge of both $decom_{id}$ and *id* such that the pairing equation is fulfilled and such that the commitment com_{id} unveils to pk_U. The operator O then only accepts if this proof is valid and if the public key has not been used before. If this succeeds the two parties engage in the creation of the initial user history.

Details of $\mathcal{R}_{\text{UR}}$

$stmt \coloneqq (\text{pk}_U,\ com_{id})$
$wit \coloneqq (decom_{id},\ id')$

$\mathcal{R}_{\text{UR}} = \{(stmt,\ wit) \mid$
- $e(\text{pk}_U, G_2) = e(G_1, id')$
- $\text{open}(com_{id},\ decom_{id},\ \text{pk}_U) = 1$

$\}$

Figure 3.10.: Zero-Knowledge relation $\mathcal{R}_{\text{UR}}$ used during user registration.

The initial contents of the user history are computed via the MPC-framework from $\mathcal{F}_{\text{PPA}}$ according to Δ which can depend on additional secret FPs which are input by the operator.

Protocol $\Pi_{\text{PUBA}}(1^\lambda, \Delta)$, Part 2

UReg:

- Input U: $(\texttt{ureg}, ssid, \texttt{in}_U)$
- Input O: $(\texttt{ureg}, ssid, fp, \texttt{in}_O)$
- Behavior:
  - U: Send $(\texttt{value})$ to $\mathcal{F}_{\text{CRS}}$ and store output $crs_{\text{ZK}}, crs_{\text{com}}$
  - U: Send $(\texttt{Retrieve}, pid_O)$ to $\mathcal{F}_{\text{BB}}$ and store output vk_O
  - U: Send $(\texttt{Retrieve}, pid_{\mathcal{TSA}})$ to $\mathcal{F}_{\text{BB}}$ and store output $\text{vk}_{\mathcal{TSA}}$
  - U: $id \xleftarrow{r} \mathbb{Z}_p$, $\text{pk}_U := id \cdot G_1$
  - U: Send $(\texttt{Register}, pid_U, \text{pk}_U)$ to $\mathcal{F}_{\text{BB}}$
  - U: $(com_{id}, decom_{id}) \leftarrow \text{commit}(crs_{\text{com}}, id)$
  - U: $stmt := (\text{pk}_U, com_{id})$
  - U: $wit := (decom_{id}, id' := id \cdot G_2)$
  - U: $\Pi \leftarrow \text{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, wit)$
  - U: $(ser^{\text{new}})^{(U)} \xleftarrow{r} \mathbb{Z}_p$
  - U: $(com^{(U)}_{ser^{\text{new}}}, decom^{(U)}_{ser^{\text{new}}}) \leftarrow \text{commit}(crs_{\text{com}}, (ser^{\text{new}})^{(U)})$
  - U: Send $(ssid\|1, pid_O, (\Pi, com_{id}, com^{(U)}_{ser^{\text{new}}}))$ to $\mathcal{F}_{\text{SMT}}$
  - O: Wait for output $(ssid\|1, pid_U, (\Pi, com_{id}, com^{(U)}_{ser^{\text{new}}}))$ from $\mathcal{F}_{\text{SMT}}$
  - O: Load $(fp, com_{fp}, decom_{fp}, \sigma_{fp})$ from $\mathsf{L}^{\texttt{ureg}}_{\text{fp}}$
  - O: Send $(\texttt{Retrieve}, pid_U)$ to $\mathcal{F}_{\text{BB}}$ and obtain output pk_U
  - O: Check that no user with public key pk_U has registered before
  - O: $stmt := (\text{pk}_U, com_{id})$
  - O: Check $\text{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \Pi) \stackrel{?}{=} 1$
  - O: Send $(ssid\|2, pid_U, (com_{fp}, \sigma_{fp}))$ to $\mathcal{F}_{\text{SMT}}$
  - U: Wait for output $(ssid\|2, pid_O, (com_{fp}, \sigma_{fp}))$ from $\mathcal{F}_{\text{SMT}}$
  - U: Check $\text{SIG.Vf}(\text{vk}_{\mathcal{TSA}}, (\texttt{ureg}, com_{fp}), \sigma_{fp}) \stackrel{?}{=} 1$
  - U: Send $(\texttt{ureg}, ssid\|1, com_{fp}, \texttt{in}_U)$ to $\mathcal{F}_{\text{PPA}}$
  - O: Send $(\texttt{ureg}, ssid\|1, fp, decom_{fp}, \texttt{in}_O)$ to $\mathcal{F}_{\text{PPA}}$
  - U: Wait for output $(ssid\|1, \mathcal{UH}, decom_{\mathcal{UH}}, \texttt{out}_U)$ from $\mathcal{F}_{\text{PPA}}$
  - O: Wait for output $(ssid\|1, com_{\mathcal{UH}}, \texttt{out}_O)$ from $\mathcal{F}_{\text{PPA}}$
  - O: $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
  - O: $(com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}) \leftarrow \text{commit}((ser^{\text{new}})^{(O)})$
  - O: $com_{ser^{\text{new}}} := com^{(O)}_{ser^{\text{new}}} \oplus com^{(U)}_{ser^{\text{new}}}$
  - O: $(com_{lin}, decom_{lin}) \leftarrow \text{commit}(0)$
  - O: $\sigma \leftarrow \text{Sign}(\text{sk}_O, (com_{\mathcal{UH}}, com_{ser^{\text{new}}}, com_{lin}, com_{id}))$
  - O: Send $(ssid\|3, pid_U, (com_{\mathcal{UH}}, (ser^{\text{new}})^{(O)}, com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}, com_{lin}, decom_{lin}, \sigma))$ to $\mathcal{F}_{\text{SMT}}$
  - U: Wait for output $(ssid\|3, pid_O, (com_{\mathcal{UH}}, (ser^{\text{new}})^{(O)}, com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}, com_{lin}, decom_{lin}, \sigma))$ from $\mathcal{F}_{\text{SMT}}$
  - U: Set $ser^{\text{new}} := (ser^{\text{new}})^{(U)} + (ser^{\text{new}})^{(O)}$
  - U: Set $com_{ser^{\text{new}}} := com^{(U)}_{ser^{\text{new}}} \oplus com^{(O)}_{ser^{\text{new}}}$
  - U: Set $decom_{ser^{\text{new}}} := decom^{(U)}_{ser^{\text{new}}} \oplus decom^{(O)}_{ser^{\text{new}}}$
  - U: Set $\text{Log}^{\text{new}} := (\mathcal{UH}, com_{\mathcal{UH}}, decom_{\mathcal{UH}}, ser^{\text{new}}, com_{ser^{\text{new}}}, decom_{ser^{\text{new}}}, 0, com_{lin}, decom_{lin}, id, com_{id}, decom_{id}, \sigma)$
  - U: Check $\Pi_{\text{Verify}}(\texttt{VfyL}, \text{Log}^{\text{new}})$ and store Log^{new}
- Output U: $(ssid, \mathcal{UH}, \texttt{out}_U)$
- Output O: $(ssid, \texttt{out}_O)$

Figure 3.9.: Protocol Π_{PUBA} part 2, user registration.

These are verified before the computation such that only FPs that were signed by $\mathcal{TSA}$ can be used for the computation.

After obtaining the initial user history this way the two parties create the initial logbook together. The principle is fairly similar in this task to all the following tasks: To ensure history freshness, the user has to carry a serial number *ser* in their logbook which is invalidated during the next interaction with the operator. However, the serial number would allow tracking of the same user through different tasks—which we want to avoid. Hence, we decided to take inspiration from the Blum coin flipping protocol [Blu81] over $\mathbb{Z}_p$ where only the user learns the outcome of the coin toss and the operator only learns that the outcome is randomly distributed over $\mathbb{Z}_p$. This is to ensure that no party picks a malicious serial number such as one that contains tracking information. Using the homomorphism of the commitment scheme COM, the operator thus can create a commitment on the *actual* serial number based only on the commitment of the user's share and their own commitment.

Using this information alongside the previously created commitments com_{lin} on the linking number (which is initially 0 as it is set only if outsourcing-triplets have been started) and $com_{\mathcal{UH}}$ and com_{id} which were obtained from the user, the operator creates a *signature*. This signature is only over commitments. Our instantiation of the signature scheme preserves the structure of the message, ensuring that this can later be used to prove in

Protocol $\Pi_{\text{PUBA}}(1^\lambda, \Delta)$, Part 3

Bookkeeping:

- Input U: $(\texttt{bookkeep}, ssid, k, \texttt{in}_U)$
- Input O: $(\texttt{bookkeep}, ssid, k, fp, \texttt{in}_O)$
- Behavior:
  - U: $(\overline{com_{\mathcal{UH}}}, \overline{decom_{\mathcal{UH}}}) \leftarrow \text{ReRand}(com_{\mathcal{UH}}, decom_{\mathcal{UH}})$
  - U: $(\overline{com_{lin}}, \overline{decom_{lin}}) \leftarrow \text{ReRand}(com_{lin}, decom_{lin})$
  - U: $(\overline{com_{id}}, \overline{decom_{id}}) \leftarrow \text{ReRand}(com_{id}, decom_{id})$
  - U: $stmt_{Val} \coloneqq (crs_{\text{com}}, \overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \text{vk}_O)$
  - U: $wit_{Val} \coloneqq (com_{\mathcal{UH}}, decom_{\mathcal{UH}}, \overline{decom_{\mathcal{UH}}}, com_{ser}, decom_{ser}, com_{lin}, decom_{lin}, \overline{decom_{lin}}, \text{pk}_U, com_{id}, decom_{id}, \overline{decom_{id}}, \sigma)$
  - U: $\Pi_{Val} \leftarrow \text{Prove}(crs_{\text{ZK}}, \mathcal{R}_{Val}, stmt_{Val}, wit_{Val})$
  - U: $(ser^{\text{new}})^{(U)} \xleftarrow{r} \mathbb{Z}_p$
  - U: $(com_{ser^{\text{new}}}^{(U)}, decom_{ser^{\text{new}}}^{(U)}) \leftarrow \text{commit}((ser^{\text{new}})^{(U)})$
  - U: Send $(\texttt{send}, ssid\|1, pid_O, (\overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \Pi_{Val}, com_{ser^{\text{new}}}^{(U)}))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - O: Wait for output $(\texttt{send}, ssid\|1, (\overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \Pi_{Val}, com_{ser^{\text{new}}}^{(U)}))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - O: $stmt_{Val} \coloneqq (crs_{\text{com}}, \overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \text{vk}_O)$
  - O: Check $\text{Vf}(crs_{\text{ZK}}, \mathcal{R}_{Val}, stmt_{Val}, \Pi_{Val}) = 1$
  - O: Check $ser \notin \mathsf{L}_{\text{SER}}$
  - O: $\mathsf{L}_{\text{SER}} \coloneqq \mathsf{L}_{\text{SER}} \cup \{ser\}$
  - O: Load $(fp, com_{fp}, decom_{fp}, \sigma_{fp})$ from $\mathsf{L}_{\text{fp}}^{\text{bookkeep}(k)}$
  - O: Send $(\texttt{respond}, ssid\|1, (com_{fp}, \sigma_{fp}))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - U: Wait for output $(\texttt{respond}, ssid\|1, (com_{fp}, \sigma_{fp}))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - U: Check $\text{SIG.Vf}(\text{vk}_{\mathcal{TSA}}, (\texttt{bookkeep}, k, com_{fp}), \sigma_{fp}) = 1$
  - U: Send $(\texttt{bookkeep}, ssid, k, \mathcal{UH}, \overline{decom_{\mathcal{UH}}}, com_{fp}, \texttt{in}_U)$ to $\mathcal{F}_{\text{PPA}}$
  - O: Send $(\texttt{bookkeep}, ssid, k, \overline{com_{\mathcal{UH}}}, fp, decom_{fp}, \texttt{in}_O)$ to $\mathcal{F}_{\text{PPA}}$
  - U: Wait for output $(ssid, \alpha, \mathbf{s}, \mathbf{a}, com_{\mathbf{a}}, decom_{\mathbf{a}}, \texttt{out}_U)$ from $\mathcal{F}_{\text{PPA}}$
  - U: **if** $\alpha = \bot \wedge \mathbf{s} = \bot$ **then**
    - Apply addition: $\mathcal{UH}^{\text{new}} \coloneqq \mathcal{UH} + \mathbf{a}$
    - $(com_{\mathcal{UH}}^{\text{new}}, decom_{\mathcal{UH}}^{\text{new}}) \leftarrow (\overline{com_{\mathcal{UH}}}, \overline{decom_{\mathcal{UH}}}) \oplus (com_{\mathbf{a}}, decom_{\mathbf{a}})$
  - U: **else**
    - Apply permutation: $\mathcal{UH}' \coloneqq \alpha(\mathcal{UH})$
    - For i from 0 to $|\mathcal{UH}| - 1$, if $\mathbf{s}[i] \neq \bot$, then set $\mathcal{UH}''[i] \coloneqq \mathbf{s}[i]$, else copy $\mathcal{UH}''[i] \coloneqq \mathcal{UH}'[i]$
    - Apply addition: $\mathcal{UH}^{\text{new}} \coloneqq \mathcal{UH}'' + \mathbf{a}$
    - $(com'_{\mathcal{UH}}, decom'_{\mathcal{UH}}) \leftarrow \text{commit}(\mathcal{UH}')$
    - $(com''_{\mathcal{UH}}, decom''_{\mathcal{UH}}) \leftarrow \text{commit}(\mathcal{UH}'')$
    - $(com_{\mathcal{UH}}^{\text{new}}, decom_{\mathcal{UH}}^{\text{new}}) \coloneqq (com''_{\mathcal{UH}}, decom''_{\mathcal{UH}}) \oplus (com_{\mathbf{a}}, decom_{\mathbf{a}})$
    - Parse $(uh_0, \ldots, uh_{m-1}) \coloneqq \mathcal{UH}$, $(uh'_0, \ldots, uh'_{m-1}) \coloneqq \mathcal{UH}'$, and $(uh''_0, \ldots, uh''_{m-1}) \coloneqq \mathcal{UH}''$
    - $stmt_{Tr} \coloneqq (crs_{\text{com}}, \overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \alpha, \mathbf{s})$
    - $wit_{Tr} \coloneqq (\overline{decom_{\mathcal{UH}}}, decom'_{\mathcal{UH}}, decom''_{\mathcal{UH}}, uh_0, \ldots, uh_{m-1}, uh'_0, \ldots, uh'_{m-1}, uh''_0, \ldots, uh''_{m-1})$
    - $\Pi_{Tr} \leftarrow \text{Prove}(crs_{\text{ZK}}, \mathcal{R}_{Tr}, stmt_{Tr}, wit_{Tr})$
    - Send $(\texttt{send}, ssid\|2, pid_O, (com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \Pi_{Tr}))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - O: Wait for output $(ssid, \alpha, \mathbf{s}, com_{\mathbf{a}}, \texttt{out}_O)$ from $\mathcal{F}_{\text{PPA}}$
  - O: **if** $\alpha \neq \bot \vee \mathbf{s} \neq \bot$ **then**
    - Wait for output $(\texttt{send}, ssid\|2, (com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \Pi_{Tr}))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
    - $stmt_{Tr} \coloneqq (crs_{\text{com}}, \overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \alpha, \mathbf{s})$
    - Check $\text{Vf}(crs_{\text{ZK}}, \mathcal{R}_{Tr}, stmt_{Tr}, \Pi_{Tr}) \stackrel{?}{=} 1$
    - $com_{\mathcal{UH}}^{\text{new}} \coloneqq com''_{\mathcal{UH}} \oplus com_{\mathbf{a}}$
  - O: **else**
    - $com_{\mathcal{UH}}^{\text{new}} \coloneqq \overline{com_{\mathcal{UH}}} \oplus com_{\mathbf{a}}$
  - O: $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
  - O: $(com_{ser^{\text{new}}}^{(O)}, decom_{ser^{\text{new}}}^{(O)}) \leftarrow \text{commit}((ser^{\text{new}})^{(O)})$
  - O: $com_{ser}^{\text{new}} \coloneqq com_{ser^{\text{new}}}^{(O)} \oplus com_{ser^{\text{new}}}^{(U)}$
  - O: $\sigma^{\text{new}} \leftarrow \text{Sign}(\text{sk}_O, (com_{\mathcal{UH}}^{\text{new}}, com_{ser}^{\text{new}}, \overline{com_{lin}}, \overline{com_{id}}))$
  - O: Send $(\texttt{respond}, ssid\|2, (com_{\mathcal{UH}}^{\text{new}}, (ser^{\text{new}})^{(O)}, com_{ser^{\text{new}}}^{(O)}, decom_{ser^{\text{new}}}^{(O)}, \sigma^{\text{new}}))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - U: Wait for output $(\texttt{respond}, ssid\|2, (com_{\mathcal{UH}}^{\text{new}}, (ser^{\text{new}})^{(O)}, com_{ser^{\text{new}}}^{(O)}, decom_{ser^{\text{new}}}^{(O)}, \sigma^{\text{new}}))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
  - U: Set $\text{Log}^{\text{new}} \coloneqq (\mathcal{UH}, com_{\mathcal{UH}}, decom_{\mathcal{UH}}, ser^{\text{new}}, com_{ser^{\text{new}}}, decom_{ser^{\text{new}}}, 0, com_{lin}, decom_{lin}, id, com_{id}, decom_{id}, \sigma)$
  - U: Check $\Pi_{\text{Verify}}(\text{VfyL}, \text{Log}^{\text{new}})$ and store Log^{new}
- Output U: $(ssid, \mathcal{UH}, \texttt{out}_U)$
- Output O: $(ssid, \texttt{out}_O)$

Figure 3.11.: Protocol Π_{PUBA} part 3, Bookkeeping.

zero-knowledge that a signature on a commitment of a given value is known. Furthermore, since the commitment itself could theoretically be used for linking when seeing it again, the operator only sees *rerandomized* versions of this commitment in future interactions alongside a zero-knowledge proof that this rerandomization is really on a commitment for which a signature made by the operator is known.

Note that all the communication happening here is *identifying* on *both* sides—even for the user. The *id* is uniquely identifiable and hence *hidden* by the user from this point onward. It is only used in the witness of zero-knowledge proofs in future interactions.

Bookkeeping The protocol for directly updating the user history is given in Fig. 3.11. Essentially, it is a wrapper around $\mathcal{F}_{\text{PPA}}$: Before accessing the hybrid functionality the user

Details of the relation $\mathcal{R}_{Val}$

$stmt \coloneqq (crs_{\text{com}}, \overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \mathsf{vk}_O)$
$wit \coloneqq (com_{\mathcal{UH}}, \overline{decom_{\mathcal{UH}}}, \overline{decom_{\mathcal{UH}}}, com_{ser}, \overline{decom_{ser}}, com_{lin}, decom_{lin}, \overline{decom_{lin}}, \mathsf{pk}_U, com_{id}, decom_{id}, \overline{decom_{id}}, \sigma)$

$\mathcal{R}_{Val} = \{(stmt, wit) \mid$
- $\mathsf{open}((com_{\mathcal{UH}} \ominus \overline{com_{\mathcal{UH}}}), (decom_{\mathcal{UH}} \ominus \overline{decom_{\mathcal{UH}}}), \mathbf{0}) = 1$
- $\mathsf{open}(com_{ser}, decom_{ser}, ser) = 1$
- $\mathsf{open}((com_{lin} \ominus \overline{com_{lin}}), (decom_{lin} \ominus \overline{decom_{lin}}), 0) = 1$
- $\mathsf{open}(com_{id}, \overline{decom_{id}}, \mathsf{pk}_U) = 1$
- $\mathsf{open}(\overline{com_{id}}, \overline{decom_{id}}, \mathsf{pk}_U) = 1$
- $\mathsf{SIG.Vf}(\mathsf{vk}_O, (com_{\mathcal{UH}}, com_{ser}, com_{lin}, com_{id}), \sigma) = 1$

$\}$

(a) Zero-Knowledge relation $\mathcal{R}_{Val}$ used to prove logbook validity.

Details of the relation $\mathcal{R}_{Tr}$

$stmt \coloneqq (crs_{\text{com}}, \overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \alpha, \mathbf{s})$
$wit \coloneqq (\overline{decom_{\mathcal{UH}}}, decom'_{\mathcal{UH}}, decom''_{\mathcal{UH}}, uh_0, \ldots, uh_{m-1}, uh'_0, \ldots, uh'_{m-1}, uh''_0, \ldots, uh''_{m-1})$

$\mathcal{R}_{Tr} = \{(stmt, wit) \mid$
- $\mathsf{open}(\overline{com_{\mathcal{UH}}}, \overline{decom_{\mathcal{UH}}}, (uh_0, \ldots, uh_{m-1})) = 1$
- $\mathsf{open}(com'_{\mathcal{UH}}, decom'_{\mathcal{UH}}, (uh'_0, \ldots, uh'_{m-1})) = 1$
- $\mathsf{open}(com''_{\mathcal{UH}}, decom''_{\mathcal{UH}}, (uh''_0, \ldots, uh''_{m-1})) = 1$
- For i from 0 to $m-1$:
  - $uh'_i = uh_{\alpha(i)}$
  - $uh''_i \coloneqq \begin{cases} \mathbf{s}[i] & \text{for } \mathbf{s}[i] \neq \perp \\ uh'_i & \text{for } \mathbf{s}[i] = \perp \end{cases}$

$\}$

(b) Zero-Knowledge relation $\mathcal{R}_{Tr}$ used to update a logbook.

Figure 3.12.: Zero-knowledge relations used for logbook manipulation.

proves to the operator that the latest input is used, and the operator proves to the user that it will input valid function parameters that were signed by $\mathcal{TSA}$. The relation $\mathcal{R}_{Val}$ for proving the former can be found in Fig. 3.12a: The user proves correct rerandomization of the user history and that they know serial, id and linking number for which they know a signature from O. For the latter—namely letting the operator prove that the FPs used for the computation are validly signed—we use an interactive protocol: O sends the commitment and the signature on the current task and the commitment to the user. Note that we do not hide from the user *which* FPs will be used—meaning that the user learns whether the same FPs have been used in previous computations—but only what those FPs *are*. Hence we do not require any form of rerandomization. The operator sends com_{fp} alongside the signature σ_{fp} directly to the user. The user then verifies the signature using the verification key $\mathsf{vk}_{\mathcal{TSA}}$ of $\mathcal{TSA}$ and only continues if this signature is valid. In that case, the user inputs the commitment to $\mathcal{F}_{\text{PPA}}$.

The operator inputs the corresponding opening information $decom_{fp}$ and the clear values of fp. The definition of $\mathcal{F}_{\text{PPA}}$ ensures that computation happens only if $decom_{fp}$ (which is an input by the operator) successfully opens com_{fp} (which is input by the user) to fp (which again is input by the operator). Assuming both unforgeability of signatures and binding of the commitment scheme, this ensures that no malicious operator can input uncertified function parameters; they either would have to forge a signature on a new commitment without knowing the signing key of $\mathcal{TSA}$ or use an existing signature but find some opening information and clear text values which open the (previously signed) com_{fp} to a different value.

After computation via $\mathcal{F}_{\text{PPA}}$, the user updates their user history. We consider this task's use case to be primarily for bookkeeping and for updating the user history, which is why we allow more complex transformations here than in Analytics. The user obtains the triple $(\alpha, \mathbf{s}, \mathbf{a})$ from $\mathcal{F}_{\text{PPA}}$, the vectors α and $\mathbf{s}$ are also learned by the operator. This triple contains a permutation α that arbitrarily permutes contents of the user history and can be the identity if no permutation is required, a set vector $\mathbf{s}$ which defines which slots of the user

history are set to new values directly and which can contain elements indicating that the old values should be used, and a private add vector $\mathbf{a}$ which defines the additive increment for each value of the user history which can be the neutral element if this entry should not be changed. The first two maps are applied to the elements in the user history directly by the user: First the contents are permuted according to α and then those slots for which an entry in $\mathbf{s}$ exists are updated to their corresponding new values. The user then proves to the operator that they updated their history correctly. For that purpose the relation $\mathcal{R}_{Tr}$ from Fig. 3.12b is used. Note that the operator never learns the actual contents of the new user history. Instead, the operator only learns a commitment on it and the permutation α and the directly updated values $\mathbf{s}$. The proof alongside the commitments on the new user history are sent to the operator. We stress that this step is optional in our protocol and can be skipped if the output of the function only contained a trivial permutation α and set vector $\mathbf{s}$ (that is, one that does not manipulate the user history). In this case both parties directly engage in the computation of the new user history instead of the user proving and the operator verifying that the updates were applied correctly.

The operator homomorphically computes the commitment of the final user history using the commitment on the permuted and updated user history from the user and the commitment on the addition vector $\mathbf{a}$ obtained from $\mathcal{F}_{\text{PPA}}$. The same technique is used to update the serial number homomorphically. This commitment is incorporated into the signature computed by the operator. The operator then sends the information required by the user to create the new logbook to the user. The user updates their logbook as in the UReg task.

Outsource The goal of the Outsource task is to *distribute* the data of the user between the operator and a proxy in such a way that 1. both parties know afterwards that the values were shared correctly, and 2. assuming P and O do not work together no information on the user data is leaked. Those two guarantees are met by the robust secret sharing protocol from Fig. 3.6 which creates shares of a given user input such that each party obtains its own additive share and a *commitment* of the *other party's* share. Additionally, each party gets their own *unveil* information for later verification.

The user uses the robust secret-sharing protocol Π_{Share}-*Share* to create robust shares of the user history $\mathcal{UH}$ which are later used for the computation, the auxiliary input $\texttt{in}_U$, and five one-time pads o_α, $\text{o}_\mathbf{s}$, $\text{o}_\mathbf{a}$, o_{OUT} and o_{UNV}. Those are also used as input for the analytics computation, as they mask the following outputs: The first three pads (o_α, $\text{o}_\mathbf{s}$ and $\text{o}_\mathbf{a}$) mask the three outputs relevant for updating the user history (namely the permutation α, the set vector $\mathbf{s}$ and the addition vector $\mathbf{a}$) later from the proxy such that given the output of $\mathcal{F}_{\text{PPA}}$ to P, only the user can reconstruct them. The fourth pad (o_{OUT}) masks the auxiliary output $\texttt{out}_U$.

During Analytics the operator only learns a commitment $com_\mathbf{a}$ of the addition vector $\mathbf{a}$. For an update that works analogous to the Bookkeeping task, the user requires the decommitment information $decom_\mathbf{a}$. Since there are commitment schemes in which access

Protocol $\Pi_{\text{PUBA}}(1^\lambda, \Delta)$, Part 4

Outsource:
- Input U: $(\texttt{outsource}, ssid, k, \texttt{in}_U)$
- Input O: $(\texttt{outsource}, ssid, k)$
- Input P: $(\texttt{outsource}, ssid, k)$
- Behavior:

U: $(otps, \langle \mathsf{Log}\rangle^O, \langle \mathsf{Log}\rangle^P) \leftarrow \Pi_{\text{Verify}}(\texttt{ShareL}, \mathsf{Log})$
U: Store $otps \coloneqq (\mathsf{o}_{\text{OUT}}, \mathsf{o}_{\text{UNV}}, \mathsf{o}_\alpha, \mathsf{o}_\mathsf{s}, \mathsf{o}_\mathsf{a})$
U: $(\overline{com_{id}}, \overline{decom_{id}}) \leftarrow \text{ReRand}(com_{id}, decom_{id})$
U: $(\mathcal{UH}^{(O)}, com^{(P)}_{\mathcal{UH}}, decom^{(O)}_{\mathcal{UH}}) \coloneqq \langle \mathcal{UH}\rangle^O$
U: $stmt \coloneqq (crs_{\text{com}}, \langle \mathcal{UH}\rangle^O, com^{(O)}_{\mathcal{UH}}, ser, \overline{com_{id}}, \mathsf{vk}_O)$
U: $wit \coloneqq (com_{\mathcal{UH}}, decom_{\mathcal{UH}}, decom^{(P)}_{\mathcal{UH}}, com_{ser}, decom_{ser}, com_{lin}, decom_{lin}, \mathsf{pk}_U, com_{id}, decom_{id}, \overline{decom_{id}}, \sigma)$
U: $\Pi \leftarrow \text{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\mathsf{0S}}, stmt, wit)$
U: $(ser^{\text{new}})^{(U)} \xleftarrow{r} \mathbb{Z}_p$
U: $(com^{(U)}_{ser^{\text{new}}}, decom^{(U)}_{ser^{\text{new}}}) \leftarrow \text{commit}((ser^{\text{new}})^{(U)})$
U: Send $(\texttt{send}, ssid\|1, pid_O, (\langle \mathsf{Log}\rangle^O, ser, \overline{com_{id}}, \Pi, com^{(U)}_{ser^{\text{new}}}))$ to $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$
U: Send $(\texttt{send}, ssid\|2, pid_P, (\langle \mathsf{Log}\rangle^P))$ to $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$
O: Wait for output $(\texttt{send}, ssid\|1, (\ldots))$ from $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$
O: Check $ser \notin \mathsf{L}_{\text{SER}}$
O: $\mathsf{L}_{\text{SER}} \coloneqq \mathsf{L}_{\text{SER}} \cup \{ser\}$
O: $(lin^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
O: $(com^{(O)}_{lin^{\text{new}}}, decom^{(O)}_{lin^{\text{new}}}) \leftarrow \text{commit}((lin^{\text{new}})^{(O)})$
O: Send $(ssid\|3, pid_P, (VerInf(\mathsf{Log})^O, com^{(O)}_{lin^{\text{new}}}))$ to $\mathcal{F}_{\text{SMT}}$
P: Wait for output $(\texttt{send}, ssid\|2, (\langle \mathsf{Log}\rangle^P, com^{(O)}_{lin^{\text{new}}}))$ from $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$ and $(ssid\|3, pid_O, (VerInf(\mathsf{Log})^O))$ from $\mathcal{F}_{\text{SMT}}$
P: Check $\Pi_{\text{Verify}}(\texttt{VfyS}, \langle \mathsf{Log}\rangle^P, VerInf(\mathsf{Log})^O) = \texttt{ok}$
P: $(lin^{\text{new}})^{(P)} \xleftarrow{r} \mathbb{Z}_p$
P: Send $(ssid\|4, pid_O, (VerInf(\mathsf{Log})^P, (lin^{\text{new}})^{(P)}))$ to $\mathcal{F}_{\text{SMT}}$
O: Wait for output $(\texttt{send}, ssid\|4, pid_P, (VerInf(\mathsf{Log})^P, (lin^{\text{new}})^{(P)}))$ from $\mathcal{F}_{\text{SMT}}$
O: $stmt \coloneqq (crs_{\text{com}}, \langle \mathcal{UH}\rangle^O, com^{(O)}_{\mathcal{UH}}, ser, \overline{com_{id}}, \mathsf{vk}_O)$
O: Check $\text{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\mathsf{0S}}, stmt, \Pi) = 1$
O: Check $\Pi_{\text{Verify}}(\texttt{VfyS}, \langle \mathsf{Log}\rangle^O, VerInf(\mathsf{Log})^P) = \texttt{ok}$
O: $lin^{\text{new}} \coloneqq (lin^{\text{new}})^{(O)} + (lin^{\text{new}})^{(P)}$
O: Add $(lin^{\text{new}}, \langle \mathsf{Log}\rangle^O)$ to $f^{(O)}_{\text{OI}}(pid_P, k)$
O: $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
O: $(com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}) \leftarrow \text{commit}((ser^{\text{new}})^{(O)})$
O: $com^{\text{new}}_{\mathcal{UH}} \coloneqq com^{(P)}_{\mathcal{UH}} \oplus com^{(O)}_{\mathcal{UH}}$
O: $com^{\text{new}}_{ser} \coloneqq com^{(U)}_{ser^{\text{new}}} \oplus com^{(O)}_{ser^{\text{new}}}$
O: $(com^{\text{new}}_{lin}, decom^{\text{new}}_{lin}) \leftarrow \text{commit}(lin^{\text{new}})$
O: $\sigma^{\text{new}} \leftarrow \text{Sign}(\mathsf{sk}_O, (com^{\text{new}}_{\mathcal{UH}}, com^{\text{new}}_{ser}, com^{\text{new}}_{lin}, \overline{com_{id}}))$
O: Send $(\texttt{respond}, ssid\|1, ((ser^{\text{new}})^{(O)}, com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}, com^{\text{new}}_{lin}, decom^{\text{new}}_{lin}, \sigma^{\text{new}}))$ to $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$
O: Send $(ssid\|5, pid_P, ((lin^{\text{new}})^{(O)}, decom^{(O)}_{lin^{\text{new}}}))$ to $\mathcal{F}_{\text{SMT}}$
P: Check $\text{open}(com^{(O)}_{lin^{\text{new}}}, decom^{(O)}_{lin^{\text{new}}}, (lin^{\text{new}})^{(O)}) = 1$
P: $lin^{\text{new}} \coloneqq (lin^{\text{new}})^{(O)} + (lin^{\text{new}})^{(P)}$
P: Add $(lin^{\text{new}}, \langle \mathsf{Log}\rangle^P)$ to $f^{(P)}_{\text{OI}}(k)$
P: Send $(\texttt{respond}, ssid\|2, (lin^{\text{new}}))$ to $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$
U: Wait for outputs $(\texttt{respond}, ssid\|1, (\ldots))$ and $(\texttt{respond}, ssid\|2, (lin^{\text{new}}))$ from $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$
U: Set $\mathsf{Log}^{\text{new}} \coloneqq (\mathcal{UH}, com^{(O)}_{\mathcal{UH}} \oplus com^{(P)}_{\mathcal{UH}}, decom^{(O)}_{\mathcal{UH}} \oplus decom^{(P)}_{\mathcal{UH}}, ser^{\text{new}}, com_{ser^{\text{new}}}, decom_{ser^{\text{new}}}, 0, com^{\text{new}}_{lin}, decom^{\text{new}}_{lin}, id, \overline{com_{id}}, \overline{decom_{id}}, \sigma^{\text{new}})$
U: Check $\Pi_{\text{Verify}}(\texttt{VfyL}, \mathsf{Log}^{\text{new}})$ and store $\mathsf{Log}^{\text{new}}$
- Output all: $(ssid, \texttt{ok})$

Figure 3.13.: Protocol Π_{PUBA} part 4, outsourcing a logbook.

to the unveil information only suffices to reconstruct the input of the commitment,[2] $\mathcal{F}_{\text{PPA}}$ *masks* the unveil information with the final OTP, o_{UNV}.

As in every interaction that requires double-spending detection, the user computes a zero-knowledge proof that the user history is valid. The relation $\mathcal{R}_{\mathsf{0S}}$ from Fig. 3.14a used for this task additionally ensures that the secret shares have been created correctly: The first line of the proof ensures that the commitment on the user history for which the user knows the signature from the operator can be homomorphically split into the two values, one of which is sent to the operator directly and one of which is sent to the proxy who then sends the commitment to the operator. Both are then used in the statement of $\mathcal{R}_{\mathsf{0S}}$. The hiding property of the commitment scheme ensures that the operator does not learn the share of the proxy, while the fact that the proxy sent the commitment to the operator

[2] Technically, the security definition of any commitment scheme makes no restriction on the amount of information regarding x stored in $decom_x$. So given any commitment scheme COM where, given $decom_x$ only, it is hard to determine x, we can create a commitment scheme COM′ which is equivalent to COM only that COM′.open sends a tuple $(decom_x, x)$. The new protocol would be as secure as the actual commitment protocol.

Details of the relation $\mathcal{R}_{\mathsf{0S}}$

$stmt \coloneqq (crs_{\mathsf{com}}, \langle \mathcal{UH} \rangle^{O}, com^{(O)}_{\mathcal{UH}}, ser, \overline{com_{id}}, \mathsf{vk}_O)$

$wit \coloneqq (com_{\mathcal{UH}}, decom_{\mathcal{UH}}, decom^{(P)}_{\mathcal{UH}}, com_{ser}, decom_{ser}, com_{lin}, decom_{lin}, \mathsf{pk}_U, com_{id}, decom_{id}, \overline{decom_{id}}, \sigma)$

$\mathcal{R}_{\mathsf{0S}} = \{(stmt, wit) \mid$
- $\mathsf{open}(com_{\mathcal{UH}} \ominus (com^{(O)}_{\mathcal{UH}} \oplus com^{(P)}_{\mathcal{UH}}), decom_{\mathcal{UH}} \ominus (decom^{(O)}_{\mathcal{UH}} \oplus decom^{(P)}_{\mathcal{UH}}), \mathbf{0}) = 1$
- $\mathsf{open}(com_{ser}, decom_{ser}, ser) = 1$
- $\mathsf{open}(com_{lin}, decom_{lin}, 0) = 1$
- $\mathsf{open}(com_{id}, decom_{id}, \mathsf{pk}_U) = 1$
- $\mathsf{open}(\overline{com_{id}}, \overline{decom_{id}}, \mathsf{pk}_U) = 1$
- $\mathsf{SIG.Vf}(\mathsf{vk}_O, (com_{\mathcal{UH}}, com_{ser}, com_{lin}, com_{id}), \sigma) = 1$

$\}$

(a) Zero-Knowledge relation $\mathcal{R}_{\mathsf{0S}}$ used when outsourcing a logbook.

Details of the relation $\mathcal{R}_{Upd}$

$stmt \coloneqq (crs_{\mathsf{com}}, \overline{com_{\mathcal{UH}}}, ser, lin, \overline{com_{id}}, \mathsf{vk}_O)$

$wit \coloneqq (com_{\mathcal{UH}}, decom_{\mathcal{UH}}, \overline{decom_{\mathcal{UH}}}, com_{ser}, decom_{ser}, com_{lin}, decom_{lin}, \mathsf{pk}_U, com_{id}, decom_{id}, \overline{decom_{id}}, \sigma)$

$\mathcal{R}_{Upd} = \{(stmt, wit) \mid$
- $\mathsf{open}((com_{\mathcal{UH}} \ominus \overline{com_{\mathcal{UH}}}), (decom_{\mathcal{UH}} \ominus \overline{decom_{\mathcal{UH}}}), \mathbf{0}) = 1$
- $\mathsf{open}(com_{ser}, decom_{ser}, ser) = 1$
- $\mathsf{open}(com_{lin}, decom_{lin}, lin) = 1$
- $\mathsf{open}(com_{id}, \overline{decom_{id}}, \mathsf{pk}_U) = 1$
- $\mathsf{open}(\overline{com_{id}}, \overline{decom_{id}}, \mathsf{pk}_U) = 1$
- $\mathsf{SIG.Vf}(\mathsf{vk}_O, (com_{\mathcal{UH}}, com_{ser}, com_{lin}, com_{id}), \sigma) = 1$

$\}$

(b) Zero-Knowledge relation $\mathcal{R}_{Upd}$ used when updating a logbook after analytics.

Figure 3.14.: Zero-knowledge relations used for outsourced analytics.

Protocol $\Pi_{\mathrm{PUBA}}(1^\lambda, \Delta)$, Part 5

Analytics:
- Input P: $(\mathtt{analyt}, ssid, k)$
- Input O: $(\mathtt{analyt}, ssid, k, fp, \mathtt{in}_O)$
- Behavior:
  - O: Load and remove the first Z_k entries $\{(lin, \langle \mathrm{Log} \rangle^O)_z\}_{z=1}^{Z_k}$ from $f^{(O)}_{\mathrm{OI}}(pid_P, k)$
  - O: Load $(fp, com_{fp}, decom_{fp}, \sigma_{fp})$ from $\mathsf{L}_{\mathsf{fp}}^{\mathtt{analyt}(k)}$
  - O: Send $(ssid \| 1, pid_P, (com_{fp}, \sigma_{fp}))$ to $\mathcal{F}_{\mathrm{SMT}}$
  - P: Wait for output $(ssid \| 1, pid_O, (com_{fp}, \sigma_{fp}))$ from $\mathcal{F}_{\mathrm{SMT}}$
  - P: Check $\mathsf{SIG.Vf}(\mathsf{vk}_{\mathcal{TSA}}, (\mathtt{analyt}, k, com_{fp}), \sigma_{fp}) \stackrel{?}{=} 1$
  - P: Load and remove the first Z_k entries $\{(lin, \langle \mathrm{Log} \rangle^P)_z\}_{z=1}^{Z_k}$ from $f^{(P)}_{\mathrm{OI}}(k)$
  - P: Send $(\mathtt{analyt}, ssid, k, com_{fp}, \{\langle \mathrm{Log} \rangle^P_z\}_{z=1}^{Z_k})$ to $\mathcal{F}_{\mathrm{PPA}}$
  - O: Send $(\mathtt{analyt}, ssid, k, fp, decom_{fp}, \{\langle \mathrm{Log} \rangle^O_z\}_{z=1}^{Z_k}, \mathtt{in}_O)$ to $\mathcal{F}_{\mathrm{PPA}}$
  - P: Wait for output $(ssid, \{\mathcal{UI}_z\}_{z=1}^{Z_k})$ from $\mathcal{F}_{\mathrm{PPA}}$
  - O: Wait for output $(ssid, \{\mathcal{UI}_z\}_{z=1}^{Z_k}, \mathtt{out}_O)$ from $\mathcal{F}_{\mathrm{PPA}}$
  - O: For all $z \in (1, \ldots, Z_k)$: set $f^{(O)}_{\mathrm{UI}}(lin_z) \coloneqq \mathcal{UI}_z \coloneqq (\alpha_z, \mathbf{s}_z, com_{\mathbf{a}_z}, ct_{\mathtt{out}_{U_z}})$
  - P: For all $z \in (1, \ldots, Z_k)$: set $f^{(P)}_{\mathrm{UI}}(lin_z) \coloneqq \mathcal{UI}_z \coloneqq (ct_{\alpha_z}, ct_{\mathbf{s}_z}, ct_{\mathbf{a}_z}, ct_{decom_{\mathbf{a}_z}}, ct_{\mathtt{out}_{U_z}})$
  - Output P: $(ssid, \mathtt{ok})$
  - Output O: $(ssid, \{\alpha_z, \mathbf{s}_z\}_{z=1}^{Z_k}, \mathtt{out}_O)$

Figure 3.15.: Protocol Π_{PUBA} part 5, analytics.

ensures that this really is the commitment that the user sent; otherwise the user could send some commitment to the operator claiming that it is the proxy's share and send a different commitment to the proxy such that the reconstructed user history contains different values than the ones stored in the user's logbook.

Since the user had to unveil the latest serial number of their logbook in order to convince the operator that indeed the latest user history was used, the protocol also includes the creation of a new logbook. Note that this logbook then has a non-zero linking number, which at the same time prohibits the user from starting a second outsourcing-triplet before finishing the first one, and stores information that can be used later by the user to fetch the results during the Update task. Other than that, the logbook generation is the same as the one used during the Bookkeeping task.

Analytics The protocol for performing analytical tasks, shown in Fig. 3.15, lets both parties fetch the values stored earlier and input them into $\mathcal{F}_{\mathrm{PPA}}$. Again, function specifics

Protocol $\Pi_{\text{PUBA}}(1^\lambda, \Delta)$, Part 6

Update:

- Input U: $(\texttt{update}, ssid)$
- Input O: $(\texttt{update}, ssid)$
- Input P: $(\texttt{update}, ssid)$
- Behavior:

U: Send $(\texttt{send}, ssid\|1, pid_P, (lin))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$

P: Wait for output $(\texttt{send}, ssid\|1, (lin))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$

P: Remove $lin \mapsto (ct_\alpha, ct_{\mathbf{s}}, ct_{\mathbf{a}}, ct_{decom_{\mathbf{a}}}, ct_{\text{out}_U})$ from $f_{\text{UI}}^{(P)}$

P: Send $(\texttt{respond}, ssid\|1, (ct_\alpha, ct_{\mathbf{s}}, ct_{\mathbf{a}}, ct_{decom_{\mathbf{a}}}, ct_{\text{out}_U}))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$

U: Wait for output $(\texttt{respond}, ssid\|1, (ct_\alpha, ct_{\mathbf{s}}, ct_{\mathbf{a}}, ct_{decom_{\mathbf{a}}}, ct_{\text{out}_U}))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$

U: $(\overline{com_{\mathcal{UH}}}, \overline{decom_{\mathcal{UH}}}) \leftarrow \text{ReRand}(com_{\mathcal{UH}}, decom_{\mathcal{UH}})$

U: $\alpha := ct_\alpha - \text{o}_\alpha$, $\mathbf{s} := ct_{\mathbf{s}} - \text{o}_{\mathbf{s}}$, $\mathbf{a} := ct_{\mathbf{a}} - \text{o}_{\mathbf{a}}$

U: $decom_{\mathbf{a}} := ct_{decom_{\mathbf{a}}} - \text{o}_{\text{UNV}}$, $\text{out}_U := ct_{\text{out}_U} - \text{o}_{\text{OUT}}$

U: $(\overline{com_{id}}, \overline{decom_{id}}) \leftarrow \text{ReRand}(com_{id}, decom_{id})$

U: $stmt_{Upd} := (crs_{\text{com}}, \overline{com_{\mathcal{UH}}}, ser, lin, \overline{com_{id}}, \text{vk}_O)$

U: $wit_{Upd} := (com_{\mathcal{UH}}, decom_{\mathcal{UH}}, \overline{decom_{\mathcal{UH}}}, com_{ser}, decom_{ser}, com_{lin}, decom_{lin}, \text{pk}_U, com_{id}, decom_{id}, \overline{decom_{id}}, \sigma)$

U: $\Pi_{Upd} \leftarrow \text{Prove}(crs_{\text{ZK}}, \mathcal{R}_{Upd}, stmt_{Upd}, wit_{Upd})$

U: $(ser^{\text{new}})^{(U)} \xleftarrow{r} \mathbb{Z}_p$

U: $(com_{ser^{\text{new}}}^{(U)}, decom_{ser^{\text{new}}}^{(U)}) \leftarrow \text{commit}((ser^{\text{new}})^{(U)})$

U: **if** $\alpha = \bot \wedge \mathbf{s} = \bot$ **then**

- Apply addition: $\mathcal{UH}^{\text{new}} := \mathcal{UH} + \mathbf{a}$
- Send $(\texttt{send}, ssid\|2, pid_O, (\overline{com_{\mathcal{UH}}}, ser, lin, \overline{com_{id}}, \Pi_{Upd}, com_{ser^{\text{new}}}^{(U)}))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$

U: **else**

- Apply permutation: $\mathcal{UH}' := \alpha(\mathcal{UH})$
- For i from 0 to $|\mathcal{UH}| - 1$, if $\mathbf{s}[i] \neq \bot$, then set $\mathcal{UH}''[i] := \mathbf{s}[i]$, else copy $\mathcal{UH}''[i] := \mathcal{UH}'[i]$
- Apply addition: $\mathcal{UH}^{\text{new}} := \mathcal{UH}'' + \mathbf{a}$
- $(com'_{\mathcal{UH}}, decom'_{\mathcal{UH}}) \leftarrow \text{commit}(\mathcal{UH}')$
- $(com''_{\mathcal{UH}}, decom''_{\mathcal{UH}}) \leftarrow \text{commit}(\mathcal{UH}'')$
- $(com_{\mathcal{UH}}^{\text{new}}, decom_{\mathcal{UH}}^{\text{new}}) := (com''_{\mathcal{UH}}, decom''_{\mathcal{UH}}) \oplus (com_{\mathbf{a}}, decom_{\mathbf{a}})$
- Parse $(uh_0, \ldots, uh_{m-1}) := \mathcal{UH}$, $(uh'_0, \ldots, uh'_{m-1}) := \mathcal{UH}'$, and $(uh''_0, \ldots, uh''_{m-1}) := \mathcal{UH}''$
- $stmt_{Tr} := (crs_{\text{com}}, \overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \alpha, \mathbf{s})$
- $wit_{Tr} := (\overline{decom_{\mathcal{UH}}}, decom'_{\mathcal{UH}}, decom''_{\mathcal{UH}}, uh_0, \ldots, uh_{m-1}, uh'_0, \ldots, uh'_{m-1}, uh''_0, \ldots, uh''_{m-1})$
- $\Pi_{Tr} \leftarrow \text{Prove}(crs_{\text{ZK}}, \mathcal{R}_{Tr}, stmt_{Tr}, wit_{Tr})$
- Send $(\texttt{send}, ssid\|2, pid_O, (\overline{com_{\mathcal{UH}}}, ser, lin, \overline{com_{id}}, \Pi_{Upd}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \Pi_{Tr}, com_{ser^{\text{new}}}^{(U)}))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$

O: Load and remove $lin \mapsto (\alpha, \mathbf{s}, com_{\mathbf{a}}, ct_{\text{out}_U})$ from $f_{\text{UI}}^{(O)}$

O: **if** $\alpha \neq \bot \vee \mathbf{s} \neq \bot$ **then**

- Wait for output $(\texttt{send}, ssid\|2, (\overline{com_{\mathcal{UH}}}, ser, lin, \overline{com_{id}}, \Pi_{Upd}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \Pi_{Tr}, com_{ser^{\text{new}}}^{(U)}))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
- $stmt_{Tr} := (crs_{\text{com}}, \overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \alpha, \mathbf{s})$
- Check $\text{Vf}(crs_{\text{ZK}}, \mathcal{R}_{Tr}, stmt_{Tr}, \Pi_{Tr}) \stackrel{?}{=} 1$
- $com_{\mathcal{UH}}^{\text{new}} := com''_{\mathcal{UH}} \oplus com_{\mathbf{a}}$

O: **else**

- Wait for output $(\texttt{send}, ssid\|2, (\overline{com_{\mathcal{UH}}}, ser, lin, \overline{com_{id}}, \Pi_{Upd}, com_{ser^{\text{new}}}^{(U)}))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$
- $com_{\mathcal{UH}}^{\text{new}} := \overline{com_{\mathcal{UH}}} \oplus com_{\mathbf{a}}$

O: $stmt_{Upd} := (crs_{\text{com}}, \overline{com_{\mathcal{UH}}}, ser, lin, \overline{com_{id}}, \text{vk}_O)$

O: Check $\text{Vf}(crs_{\text{ZK}}, \mathcal{R}_{Upd}, stmt_{Upd}, \Pi_{Upd}) \stackrel{?}{=} 1$

O: Check $ser \notin \mathsf{L}_{\text{SER}}$

O: $\mathsf{L}_{\text{SER}} := \mathsf{L}_{\text{SER}} \cup \{ser\}$

O: $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$

O: $(com_{ser^{\text{new}}}^{(O)}, decom_{ser^{\text{new}}}^{(O)}) \leftarrow \text{commit}((ser^{\text{new}})^{(O)})$

O: $com_{ser}^{\text{new}} := com_{ser^{\text{new}}}^{(O)} \oplus com_{ser^{\text{new}}}^{(U)}$

O: $(com_{lin}^{\text{new}}, decom_{lin}^{\text{new}}) \leftarrow \text{commit}(0)$

O: $\sigma^{\text{new}} \leftarrow \text{Sign}(\text{sk}_O, (com_{\mathcal{UH}}^{\text{new}}, com_{ser}^{\text{new}}, com_{lin}^{\text{new}}, \overline{com_{id}}))$

O: Send $(\texttt{respond}, ssid\|2, (com_{\mathbf{a}}, com_{\mathcal{UH}}^{\text{new}}, (ser^{\text{new}})^{(O)}, com_{ser^{\text{new}}}^{(O)}, decom_{ser^{\text{new}}}^{(O)}, com_{lin}^{\text{new}}, decom_{lin}^{\text{new}}, \sigma^{\text{new}}, ct_{\text{out}_U}))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$

U: Wait for output $(\texttt{respond}, ssid\|2, (com_{\mathbf{a}}, com_{\mathcal{UH}}^{\text{new}}, (ser^{\text{new}})^{(O)}, com_{ser^{\text{new}}}^{(O)}, decom_{ser^{\text{new}}}^{(O)}, com_{lin}^{\text{new}}, decom_{lin}^{\text{new}}, \sigma^{\text{new}}, ct_{\text{out}_U}))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$

U: Check $\text{open}(com_{\mathbf{a}}, decom_{\mathbf{a}}, \mathbf{a}) \stackrel{?}{=} 1$

U: Set $ser^{\text{new}} := (ser^{\text{new}})^{(U)} + (ser^{\text{new}})^{(O)}$

U: Set $com_{ser^{\text{new}}} := com_{ser^{\text{new}}}^{(U)} \oplus com_{ser^{\text{new}}}^{(O)}$

U: Set $decom_{ser^{\text{new}}} := decom_{ser^{\text{new}}}^{(U)} \oplus decom_{ser^{\text{new}}}^{(O)}$

U: Set $\text{Log}^{\text{new}} := (\mathcal{UH}^{\text{new}}, com_{\mathcal{UH}}^{\text{new}}, (\overline{decom_{\mathcal{UH}}} \oplus decom_{\mathbf{a}}), ser^{\text{new}}, com_{ser^{\text{new}}}, decom_{ser^{\text{new}}}, 0, com_{lin}^{\text{new}}, decom_{lin}^{\text{new}}, id, \overline{com_{id}}, \overline{decom_{id}}, \sigma^{\text{new}})$

U: Check $\Pi_{\text{Verify}}(\text{VfyL}, \text{Log}^{\text{new}})$ and store Log^{new}

- Output U: $(ssid, \alpha, \mathbf{s}, \mathbf{a}, \text{out}_U)$
- Output P: $(ssid, \text{ok})$
- Output O: $(ssid, \alpha, \mathbf{s})$

Figure 3.16.: Protocol Π_{PUBA} part 6, updating a logbook after analytics.

are input by the operator as *fp* and the input is verified beforehand using the same basic technique as in the Bookkeeping task—but this time against the proxy.

After receiving output from $\mathcal{F}_{\text{PPA}}$, both parties store the results for later use.

Update The Update protocol, shown in Fig. 3.16, contains two steps which do not necessarily have to be executed at once. In the first step the user only requests the data P has stored by sending the linking number. In the second step the user requests the data from O and additionally computes a new logbook and proves that the data in there has

been applied correctly. This proof employs the relation $\mathcal{R}_{Upd}$ from Fig. 3.14b which is reminiscent of the proof used during Π_{PUBA}-*Bookkeeping*, but also proves that the *linking number* is correct. Additionally, in case of a non-trivial permutation or direct update the user uses the relation $\mathcal{R}_{Tr}$ in order to prove that the permutation was applied correctly.

3.4.9. Formal Security Statement

As already mentioned in Section 3.3, we prove security of our protocol Π_{PUBA} in the UC framework by showing indistinguishability against the ideal functionality $\mathcal{F}_{\text{PUBA}}$. In this section we elaborate on the achieved security level.

Security Theorem We can now state our main security theorem:

Theorem 3.4.1. *Let $\mathcal{Z}$ be any PPT-environment that corrupts a subset of parties that does not contain the operator and the proxy at the same time and that does not contain the TSA, and let the building blocks be instantiated as described in Section 3.4.2. Then it holds that:*

$$\Pi_{\text{PUBA}}{}^{(\mathcal{F}_{\text{CRS}},\mathcal{F}_{\text{SMT}},\mathcal{F}_{\text{SMT}}^{\text{os-auth}},\mathcal{F}_{\text{BB}},\mathcal{F}_{\text{PPA}})} \geq_{UC} \mathcal{F}_{\text{PUBA}}$$

This means that the protocol Π_{PUBA} is at least as secure as $\mathcal{F}_{\text{PUBA}}$.

We defer the formal proof of this theorem to Section A.3.

3.5. Application Examples

PUBA can be used for a variety of different applications. Here we sketch two privacy-preserving applications for PUBA, namely fraud detection for mobile payments and a targeted advertising system.

3.5.1. Fraud Detection for Mobile Payments

In the European Central Bank's latest report on card fraud [Eur18] published in 2018, the total value of fraudulent transactions at points-of-sale in 2016 amounted to about 342 Million Euro. This number sounds high, yet it only amounts to 0.008% of the overall card transaction value. This small ratio is achieved mainly thanks to the use of "strong authentication" methods like Chip&Pin as well as fraud detection mechanisms required by the 2nd European Payment Services Directive (PSD2).

We define a privacy-preserving mobile payment system that includes fraud detection capabilities. The operator in this system is the bank that offers the mobile payments service to its customers. These are the users in our system who interact using a smartphone app.

Fraud detection consists in monitoring a customer's transactions for anomalies or typical fraud patterns. This can be done based on simple rules or sophisticated machine learning algorithms. Due to the real-time requirements, the combination of fraud detection with privacy for mobile payments is particularly challenging. To this end, we consider the following two-tier mechanism: We force the user to perform a more complex machine-learning-based fraud detection with the operator if some threshold of payment transactions has been reached, resulting in some risk level, and a simple but faster rule-based fraud detection during each payment at a point-of-sale. The latter takes the risk level into account and decides whether the current transaction is accepted or declined.

Each time the user conducts a payment at a point-of-sale, a new transaction is created and stored in the user history. We assume that a transaction record t consists of the following data encoded as vector of $\mathbb{Z}_p$ elements:

$$t \coloneqq (\mathsf{acc}, \mathsf{ts}, \mathsf{loc}, \mathsf{type}, \mathsf{tval}),$$

where acc is one iff the transaction has been accepted, ts is a timestamp, loc indicates the geographic location the transaction took place, type describes the type of shop (e.g., grocery store, jewelry store, etc.), and tval is the transaction value. We stress that this is only an example, adding other attributes is pretty straightforward.

The user history contains the latest T transaction records, the user's balance *bal* and some important additional information to support the fraud detection mechanisms. These additional information include the account's current risk level *rsk*, a maximum value *max* a single transaction can have, and a limit *rem* on the number of payment transactions a user can perform before the complex fraud detection mechanism has to be run. Note that the latter two values depend on the current risk level.

Thus, the user history has the following form:

$$\mathcal{UH} \coloneqq (t_1^1, \ldots, t_5^1, \ \ldots \ , t_1^T, \ldots, t_5^T, \mathit{rsk}, \mathit{rem}, \mathit{max}, \mathit{bal})$$

The first $5T$ slots store the last T transactions (each transaction t^i requires 5 slots), t^1 is the most recent transaction. The user history has a total length of $5T + 4$ entries.

We now describe the individual tasks a user can perform. The details of the function Δ_{mpayment} can be found in Fig. 3.17. We estimate the application's performance in Section 3.6.4.

Registration During registration the user obtains a user history with an empty balance and no stored transactions using the UReg task, which calls the function Δ_{mpayment} with input message `ureg`. Both parties input the initial values for the risk level, the remaining number of transactions and the maximum transaction value, i.e., (*rsk*, *rem*, *max*). The values *rem* and *max* can either be fixed constants or depend on the risk level. Alongside empty transactions, those are written into the new user history.

Δ_{mpayment}

Registration:
$(\texttt{ureg},\ fp \coloneqq \bot,\ \texttt{in}_U \coloneqq (rsk,\ rem,\ max),\ \texttt{in}_O \coloneqq (rsk,\ rem,\ max))$
If $\texttt{in}_U \neq \texttt{in}_O$, abort
Set $\mathcal{UH} \coloneqq (\bot, \ldots, \bot, rsk, rem, max, 0)$
Return $(\mathcal{UH},\ \texttt{out}_U \coloneqq \text{ok},\ \texttt{out}_O \coloneqq \text{ok})$

Top-Up:
$(\texttt{bookkeep},\ fp \coloneqq \bot,\ k \coloneqq 1,\ \mathcal{UH},\ \texttt{in}_U \coloneqq \text{val},\ \texttt{in}_O \coloneqq \text{val})$
If $\texttt{in}_U \neq \texttt{in}_O$, abort
$\mathbf{a} \coloneqq (0, \ldots, 0, \text{val})$
Return $(\alpha \coloneqq \left(\begin{smallmatrix} 0 & \ldots & m-1 \\ 0 & \ldots & m-1 \end{smallmatrix}\right),\ \mathbf{s} \coloneqq (\bot, \ldots, \bot),\ \mathbf{a},\ \texttt{out}_U \coloneqq \text{ok},\ \texttt{out}_O \coloneqq \text{ok})$

Payment:
$(\texttt{bookkeep},\ fp \coloneqq \bot,\ k \coloneqq 2,\ \mathcal{UH},\ \texttt{in}_U \coloneqq (\text{ts},\ \text{loc},\ \text{type},\ \text{tval}),\ \texttt{in}_O \coloneqq (\text{ts},\ \text{loc},\ \text{type},\ \text{tval}))$
If $\texttt{in}_U \neq \texttt{in}_O$, abort
Parse $(t_1^1, \ldots, t_5^1,\ \ldots,\ t_1^T, \ldots, t_5^T, rsk, rem, max, bal) \coloneqq \mathcal{UH}$
If $bal <$ tval, then return $(\alpha \coloneqq \left(\begin{smallmatrix} 0 & \ldots & m-1 \\ 0 & \ldots & m-1 \end{smallmatrix}\right), \mathbf{s} \coloneqq (\bot, \ldots, \bot), \mathbf{a} \coloneqq (0, \ldots, 0), \texttt{out}_U \coloneqq \bot, \texttt{out}_O \coloneqq \bot)$
$\text{acc} \coloneqq f_{\text{fdsimple}}(\mathcal{UH}, (\text{ts}, \text{loc}, \text{type}, \text{tval}))$
$(t_1^{\text{new}}, \ldots, t_5^{\text{new}}) \coloneqq (\text{acc}, \text{ts}, \text{loc}, \text{type}, \text{tval})$
If acc = 1,
  then $bal^{\text{diff}} \coloneqq \text{tval}$,
  else $bal^{\text{diff}} \coloneqq 0$
$\alpha \coloneqq \left(\begin{smallmatrix} 0 & \ldots & 4 & 5 & \ldots & 9 & \ldots & 5T-5 & \ldots & 5T-1 & 5T & \ldots & 5T+3 \\ 5T-5 & \ldots & 5T-1 & 0 & \ldots & 4 & \ldots & 5T-10 & \ldots & 5T-6 & 5T & \ldots & 5T+3 \end{smallmatrix}\right)$
$\mathbf{s} \coloneqq (t_1^{\text{new}}, \ldots, t_5^{\text{new}}, \bot, \ldots, \bot)$
$\mathbf{a} \coloneqq (0, \ldots, 0, 0, -1, 0, -bal^{\text{diff}})$
Return $(\alpha,\ \mathbf{s},\ \mathbf{a},\ \texttt{out}_U \coloneqq \text{acc},\ \texttt{out}_O \coloneqq \text{acc})$

Risk Calculation:
$(\texttt{analyt},\ fp,\ k \coloneqq 3,\ (\mathcal{UH},\ \texttt{in}_U \coloneqq \bot),\ \texttt{in}_O \coloneqq \bot)$
Parse $(t_1^1,\ \ldots,\ t_5^1,\ \ldots,\ t_1^T,\ \ldots,\ t_5^T,\ rsk,\ rem,\ max,\ bal) \coloneqq \mathcal{UH}$
$(rsk^{\text{new}},\ rem^{\text{new}},\ max^{\text{new}}) \coloneqq f_{\text{fdcomp}}(fp, \mathcal{UH})$
$\mathbf{s} \coloneqq (\bot,\ \ldots,\ \bot,\ rsk^{\text{new}},\ \bot,\ max^{\text{new}},\ \bot)$
$\mathbf{a} \coloneqq (0,\ \ldots,\ 0,\ 0,\ -rem + rem^{\text{new}},\ 0,\ 0)$
Return $((\alpha \coloneqq \left(\begin{smallmatrix} 0 & \ldots & m-1 \\ 0 & \ldots & m-1 \end{smallmatrix}\right),\ \mathbf{s},\ \mathbf{a},\ \texttt{out}_U \coloneqq \text{ok}),\ \texttt{out}_O \coloneqq \text{ok})$

Figure 3.17.: Instantiation of Δ for privacy-preserving mobile payments with fraud detection.

Top-Up Our mobile payments service uses a prepaid model; it is important that users can top up their balance. We propose a general method where the user and the bank agree on the amount that should be topped up and leave the actual transfer of money to the implementation method, such as anonymously depositing money at an ATM or making a transfer from the normal bank account (which would be identifying). The user invokes a Bookkeeping task where both parties input the amount to be deposited, which is added to the user's balance. In our example, a top-up transaction is not recorded in the user's transaction history (although this might be reasonable).

Payment (with simple fraud detection) To issue a payment, a user communicates with a point-of-sale that has a communication channel with the bank and that forwards all messages between the user and the bank. The payment is done via the Bookkeeping task where both parties input all transaction details excluding the acceptance bit, i.e., timestamp, location, type of shop and transaction value. The function aborts if the transaction value exceeds the account's balance. Otherwise, the simple rule-based fraud detection mechanism f_{fdsimple} is executed to decide whether the transaction is accepted or not based

on the last T transactions, the risk level *rsk*, the remaining number of transactions *rem*, the maximum transaction value *max*, and the details of the current transaction. In our example implementation in Section 3.6.4 we verify that the following conditions are all satisfied: 1. tval $\leq$ *max* (the transaction value does not exceed the maximum allowed amount) 2. $rem > 0$ (the number of payment transactions the user can perform before the complex fraud detection mechanism has to be run has not exceeded its limit). Of course, additional checks could be included: The transaction could be denied if the risk level is medium but there are more than three transactions within a 10 minute period, or if two consecutive transactions differ in their location so much that no user could have possibly traveled that far in such a short time period.

This simple mechanism already provides some fraud protection, but is lightweight enough to be computed by the user's resource-constrained device. As those can be public, the user can evaluate the rule-based fraud detection themselves and provide the point-of-sale with a zero-knowledge proof that they evaluated the mechanism correctly based on their logbook. This can significantly speed up the payment process compared to an MPC-based computation, assuming the rules are simple and efficiently compatible with the zero-knowledge proof system.

Only if the transaction is accepted it is physically executed and the user's balance gets updated accordingly. More specifically, the three output maps of Δ look as follows: The permutation shifts all past transactions to the right to make room for the new transaction, which is written into the user history with the direct update. This is done because the user history only stores the last T transactions to hide the total number of transactions. The additive increment then subtracts one from the number of remaining transactions *rem* and subtracts the transaction value tval from the balance *bal* iff the transaction was accepted. The risk level and the maximum transaction value stay the same.

More precisely, the new transaction is assembled as $(t_1^{\text{new}}, \ldots, t_5^{\text{new}}) = (\text{acc}, \text{ts}, \text{loc}, \text{type}, \text{tval})$ and the new user history then looks as follows after applying the three maps: $\mathcal{UH}^{\text{new}} \coloneqq (t_1^{\text{new}}, \ldots, t_5^{\text{new}}, t_1^1, \ldots, t_5^1, \ldots, t_1^{T-1}, \ldots, t_5^{T-1}, rsk, rem - 1, max, bal^{\text{new}})$.

Risk Calculation (with complex fraud detection) Each time the user executes a payment transaction, the counter for the number of remaining transactions decreases by one. When this counter reaches zero, it forces the user to participate in this task, where the risk level gets updated and a more sophisticated fraud detection algorithm is executed. By choosing a suitable value for the initial value of that counter, we can ensure that users regularly participate in the complex fraud detection mechanism. As we assume this complex fraud detection mechanism to be based on machine learning, this might result in considerable computational effort. Therefore, the Analytics task is used. The operator inputs its FPs *fp* into Analytics. Note that, as usually, these FPs were verified by the TSA to pose no privacy-risk for the user and yet are not learned by the user. The fraud detection mechanism then computes the user's new risk level *rsk* along with a new maximum number of transactions *rem*, and maximum transaction value *max*. These new values are then stored in the user history. More specifically, only the direct update and additive increment

are needed. The direct update sets the new values for the risk level and the new maximum transaction value at the corresponding slots and overwrites the old values in the process. Since the outsourcing triple is non-blocking regarding Bookkeeping operations, we have to take into account that the value of the remaining number of transaction may have changed since Outsource was called. Therefore, the additive increment adds the difference between the old remaining number of transactions (from the point when Outsource was called) and the new value to the corresponding slot.

Since this is an outsourced analytics computation, we need to assume the existence of a proxy that assists the user in this computationally expensive computation. Since today's banks already use external companies to help them with credit card fraud detection, such a company could act as the proxy.

3.5.2. Targeted Advertising System

We now briefly sketch a targeted advertising system which can optionally be used as an extension for loyalty systems. For their cooperation, users are rewarded with vouchers targeted at their purchase behavior. The central idea is that the user's purchases are stored in the user history. From time to time, users submit their purchase history to the operator, who analyzes it together with the histories of several other users. The user is rewarded with a voucher targeted at the user's probable interests and is displayed alongside a suitable ad in the smartphone app.

We now assume that the operator acts as a conglomeration of supermarket chains and further participating shops. In the following, we describe how to use PUBA in this scenario.

Registration Upon registration with the UReg task, the user obtains an empty user history. Each slot in the user history represents a product category, e.g., "vegetables", "candy", or "fast food". The user history tracks the amount of money spent in each category.

Checkout When purchasing goods at a participating store, the user updates the purchase history using the Bookkeeping task. The amount of money spent in each category is calculated and added to the corresponding slots in the user history.

Analytics The Analytics task lets the user provide data for analytical purposes. We assume the operator has an analytical function (for example for marketing analyses) which takes some FPs and multiple user histories as input and assigns to each user history a class that describes the most likely interests of the corresponding user. The user is rewarded with a voucher that matches this class and that can be redeemed at a participating shop. Additionally, the user gets a matching advertisement. For example, if the analysis reveals that the user likes chocolate, the user obtains advertisements for a new kind of chocolate

and a voucher for a 10% discount on chocolate. The user histories of the participating users remain unchanged.

If the targeted advertising system is interconnected with a loyalty system, the user could also earn loyalty points instead of vouchers.

3.6. Implementation

We evaluated the practicality of PUBA by measuring the execution times of a practical implementation. Since network communication depends on various external factors, we omitted communication times and only measured local computation. The implementation is available at `https://github.com/kastel-security/puba`.

3.6.1. Bookkeeping

Evaluation of the user side is done on a Nexus 5X smartphone released 2015 featuring a Snapdragon 808 with 2×1.8 GHz + 4×1.4 GHz running Android 8.1.0 (Phone 1) and a Galaxy S8 smartphone released 2017 featuring an Exynos 8895 with 4×2.3 GHz + 4×1.7 GHz running Android 9 (Phone 2). We executed the code for operator and proxies on much more powerful servers, equipped with an AMD Ryzen 9 3900X with 12×3.8 GHz. In all cases our implementation makes use of 6 threads to speed up cryptographic operations.

We implemented our protocol in C++17, employing the open-source library RELIC toolkit v0.5.0 [AG20] for group operations. The required building blocks were instantiated as suggested in Section 3.4.2: for *signatures* we use the scheme from [Abe+11], for *commitments* we implemented [Abe+15], and for the NIZKPoK scheme we use the method from [GS08; EG14]. Our building blocks are instantiated over the pairing-friendly Barreto–Naehrig Curves Fp254BNb and Fp254n2BNb [Ara+11; BN06].

We averaged over 50 executions of each protocol task for user history sizes of 10, 100, 200, 400, 600, 800 and 1000 in order to get representative results. For Bookkeeping we implemented both the three-stage update comprising a permutation of user history, setting values, and adding values, as well as a simplified version where no permutation and direct update are done. The concrete choice of permutation has no impact on performance, whereas performance improves slightly the more entries are set. Thus, to provide a good lower bound, we performed a cyclic shift and then set a single entry. Table 3.1 shows the results using a server for operator and proxy and Table 3.2 using a smartphone for the user side.

When only performing addition during Bookkeeping (and no permutation/setting of entries), it needs to communicate ≈4 kB regardless of the number of entries in the user history, and takes between ≈310 ms and ≈440 ms on Phone 1 and ≈460 ms on Phone 2, while taking 14 ms for the operator.

$\lvert\mathcal{UH}\rvert$	UReg		Bookkeeping		Outsource			Update	
	$\sum$	ZK	$\sum$	ZK	$\sum$	ZK	P	$\sum$	ZK
10	6	3	22	16	20	10	4	20	12
100	6	3	57	51	30	10	15	54	44
200	6	3	96	90	42	10	26	93	80
400	6	3	176	170	60	10	44	172	152
600	7	3	254	248	79	10	65	251	225
800	7	3	332	327	97	10	84	331	299
1000	7	3	411	405	117	10	105	408	370

Table 3.1.: Running times for operator/proxy in ms. $\sum$ denotes the total operator running time, ZK the portion of that spent verifying the zero-knowledge proof and P denotes the running time of the proxy during Outsource. Proxy running time during Update is <1 ms for all User History sizes.

Overall, as can be seen in Table 3.3, even for user history sizes where computation time on a smartphone exceeds 10 s, less than 300 kB of data need to be communicated. Thus, even when using mobile data, communication times will mostly depend on network latency and in general be relatively short.

3.6.2. Analytics Computation

We implemented logistic regression inference using MP-SPDZ [Kel20], which allows for benchmarking across a range of security models and protocols. As the cleartext modulus of the used curve is not compatible with the implementation of homomorphic encryption in MP-SPDZ, we restrict ourselves to protocols based on oblivious transfer with malicious security. For this we use MASCOT [KOS16]. As MP-SPDZ already implements logistic regression, we only had to choose the number of features and whether to approximate the sigmoid function for faster computation. For the former, we ran inference for 10, 20, 50, 100, and 1000 features, and for the latter benchmarked both the established sigmoid function and the three-piece approximation [MR18]. The latter has been found to deliver good results while being much simpler to compute in the context of secure computation. This is because the restriction to three linear pieces only requires two comparisons and oblivious selections instead of exponentiation and logarithm. To define the computation domain, we used the order of the 254-bit prime field Weierstrass curve. This allows for a smooth integration with our zero-knowledge proof.

Table 3.4 shows our end-to-end timings when running on AWS c5.9xlarge and m4.large instances. The LAN times refers to the colocated setting. We simulate a WAN setting by adding a 100 ms roundtrip delay and a bandwidth restriction of 10 Mbit/s. We only use one thread and about 300 MB RAM for malicious security.

$\lvert\mathcal{UH}\rvert$	Phone	UReg				Bookkeeping				Outsource					Update			
		$\sum$	O	ZK	Val	$\sum$	O	ZK	Val	$\sum$	PC	O	ZK	Val	$\sum$	O	ZK	Val
10	1	154	97	78	57	616	556	494	60	551	189	298	193	64	528	461	400	67
	2	181	99	73	81	969	881	788	87	749	244	409	272	87	809	719	622	90
100	1	155	100	78	56	1559	1501	1432	58	1247	818	369	181	63	1541	1439	1319	107
	2	176	96	71	80	2200	2116	2033	84	1750	1180	477	240	89	2102	2008	1842	92
200	1	223	116	86	84	2725	2616	2542	87	2088	1540	499	204	92	2868	2723	2484	124
	2	175	96	70	79	3465	3382	3300	82	2959	2242	625	235	86	3353	3265	3011	90
400	1	241	122	91	118	5649	5530	5441	119	4309	3373	804	221	128	5633	5502	5015	130
	2	178	96	71	82	6302	6217	6136	85	5603	4508	986	249	88	6422	6331	5902	93
600	1	240	120	88	120	9226	9103	9003	121	7073	5773	1182	252	126	9210	9079	8406	129
	2	178	96	70	84	9695	9612	9519	84	8228	6801	1336	265	88	9978	9885	9269	92
800	1	242	121	90	122	11 899	11 775	11 672	125	9386	7763	1460	253	129	12 100	11 966	11 104	131
	2	186	99	74	87	13 130	13 046	12 950	85	10 778	9020	1656	265	90	13 402	13 311	12 462	93
1000	1	245	121	90	123	14 667	14 539	14 443	129	11 440	9594	1708	256	131	14 940	14 807	13 763	130
	2	189	100	74	92	16 383	16 290	16 213	91	13 447	11 377	1987	269	91	16 642	16 538	15 471	94

Table 3.2.: Running time on user devices in ms. $\sum$ denotes the total user running time, **O** denotes the online running time, **ZK** denotes the portion of that spent creating the zero-knowledge proofs, **Val** the time spent validating the new User History (not included in the online time) and **PC** precomputation time (also not included in the online time).

$\lvert\mathcal{UH}\rvert$:	10	100	200	400	600	800	1000
UReg	1.6	1.6	1.6	1.6	1.6	1.6	1.6
Bookkeeping	5.8	11.8	18.4	31.6	44.8	58.0	71.2
(only add)	4.1	4.1	4.1	4.1	4.1	4.1	4.1
Outsource	8.9	34.1	62.0	118.0	174.0	229.9	285.9
Update	7.0	22.6	40.0	74.8	109.6	144.4	170.2

Table 3.3.: Data exchanged in relation to user history size in kB.

No. features	Precise sigmoid				Approximate sigmoid			
	S-LAN	LAN	WAN	MB	S-LAN	LAN	WAN	MB
10	3.60	5.06	94.86	591.74	0.32	0.59	11.61	69.71
20	3.62	5.07	95.12	593.29	0.33	0.60	11.93	71.26
50	3.65	5.11	95.84	597.94	0.35	0.64	12.58	75.91
100	3.68	5.19	96.90	605.68	0.38	0.71	13.53	83.66
1000	4.28	6.47	116.10	745.10	0.97	1.87	32.82	223.07

Table 3.4.: Time and communication for logistic regression with two parties. Times are given in seconds (s). "S-LAN" refers to c5.9xlarge instances, otherwise we use m4.large.

At the time of writing the paper, the spot price in US East was USD 0.10 and 1.53 for m4.large and c5.9xlarge, respectively. This results in a cost per computation ranging from USD 1.6×10^{-5} to USD 0.0032.

3.6.3. Discussion

Our results show that, even on weak hardware compared to modern smartphones, for moderately sized user histories our protocol runs fast enough for a smooth user experience: A Bookkeeping—which will be executed most frequently—runs in less than 3 s even including typical network latency for user histories with 100 entries. When Bookkeeping can be performed without the need of permuting or setting user history entries, it runs in well under 1 s and can even support much larger user history sizes. Outsource needs to transmit more data but is also used less frequently, and a large part of the necessary computation is independent of the current user history state (i.e., creating commitments to shares of one-time pads and the random shares for one of the two parties) and can thus be done in the background in advance. This allows Outsource to have very short online running time even for large user history sizes. Update on the other hand takes about the same time as Bookkeeping. Thus, the main limiting factor for the size of the user history is the time needed for Bookkeeping. This can be partially mitigated if the need of permuting/setting values arises only sparingly or if only parts of the user history are affected. Then the whole user history can be split into multiple parts where permuting/setting is done on some parts and a simple additive update is performed on

the other parts. Our evaluation also shows that Phone 2 was consistently slower than Phone 1, even though the stronger processor would suggest otherwise. We are not certain about the cause of this. Possible explanations are the introduction of new battery saving mechanisms in Android 9, or the fact that Phone 1 has the stock google version of Android whereas Phone 2 has vendor-specific modifications.

3.6.4. Performance of Fraud Detection

We evaluated our fraud detection application from Section 3.5.1 with a user history containing 100 elements, corresponding to a scenario where the last $T = 19$ transactions are taken into account for analysis (5 entries per transaction plus previous risk level, number of remaining transactions, transaction limit and balance). We consider these numbers to be realistic and discuss them in Section A.1.3.

Since Registration only needs to initialize the default risk level, number of remaining transactions and transaction limit, this is independent of the user history size and takes the same time and data volume as the UReg task shown in Table 3.2 (i.e., 160 ms for Phone 1 and 190 ms for Phone 2 combined user plus operator computation time excluding communication time and 1.6 kB data). Similarly, the Top-Up task does not require any special computation and thus the results for Bookkeeping with only addition holds for this task (i.e., 330 - 450 ms for Phone 1 and ≈490 ms for Phone 2 combined user plus operator computation time excluding communication time and 4.1 kB data).

Fraud detection happens in two stages: A more complex machine-learning-based risk assessment is performed regularly, which results in a risk score, a limit for individual payments and a limit for the number of transactions before it has to be executed again. During each payment, simpler rules based on the result of the risk assessment are checked. In our implementation we verified the following two rules: 1. the transaction value is less than or equal to the maximum allowed transaction amount 2. the number of payment transactions the user can perform before the complex fraud detection mechanism has to be run has not exceeded its limit. Additionally, it is verified that the current balance is sufficient for the transaction. To do so, we use bulletproof rangeproofs [Bün+18] to let the user prove that they evaluated the simple rules correctly. These additional proofs take ≈200 ms on Phone 1 and ≈185 ms on Phone 2, and 17 ms for the operator, independent of the user history size. Thus, for 100 entries (corresponding to the last $T = 19$ transactions), this task takes a total of 1.85 s/2.5 s (for Phone 1/2 resp.) on the user side and 80 ms on the operator side (excluding the communication time to transmit 22 kB of data). This shows that, when storing 19 previous transactions, a transaction can be performed in under 3 s even when taking the communication time into account. Increasing the number of previous transactions increases the computation time on average by ≈1.5 s/≈1.6 s for Phone 1/2 resp. and 40 ms for the operator per 20 additional transactions.

For the outsourced risk calculation we used logistic regression on 100 features using approximate sigmoid calculation and active security for the complex fraud detection mechanism. Our results suggest that the whole outsourced risk calculation process can be

completed in well under 5 s: ≈0.7 s for outsourcing, ≈0.4 s for the logistic regression and ≈1.3 s for the update, plus communication time between the user and operator/proxy.

4. POBA: Storing data on operator servers

This chapter is based on "POBA: Privacy-Preserving Operator-Side Bookkeeping and Analytics" [Fau+25c; Fau+25a] and in most parts taken verbatim from there, except Section 4.6, which is original work new in this thesis and Section 4.5, for which new benchmarks with a new implementation were done.

My Contribution In the paper, I was the main contributor of the system modelling and protocol design, and did all formal proofs. For this thesis, I additionally re-implemented the protocol and redid most of the benchmarks.

Organization of this chapter We first provide an introduction and the related work in Section 4.1. Then, we discuss the desired functionality and involved parties and present the ideal functionality $\mathcal{F}_{\text{POBA}}$ in Section 4.2. Section 4.3 contains our protocol Π_{POBA} for privacy-preserving bookkeeping and analytics with operator-side data storage. Section 4.4 contains details on how to use POBA in example applications. In Section 4.5 we present our prototype implementation and benchmark results. Finally, Section 4.6 discusses the steps needed to upgrade the protocol from security against semi-honest operators to full malicious security.

4.1. Introduction

Privacy-preserving data analysis is now more important than ever: Companies generate vast amounts of data that they would like or need to analyze. A number of studies have shown that individuals are willing to share their data with companies if there is a potential benefit to them.[1] However, it is crucial to ensure that this data is processed in a way that protects individuals' privacy. Furthermore, the storage of large amounts of sensitive data for analysis is also a source of concern: There are recurrent cases where cybercriminals

[1] https://venturebeat.com/business/were-giving-away-more-personal-data-than-ever-despite-growing-risks/

infiltrate company networks and steal large amounts of user-specific data.[2] These leaks have a negative impact on companies as well as the individuals whose data gets abused. To prevent these data leaks, it would make sense to not collect sensitive user data in the first place. This is not always an option, however, either because of laws/regulations (anti-money laundering, etc.), or the companies need the data for their business model (for usage-based fees, targeted ads, etc.), for behavior-dependent pricing (behavior-based car insurance, etc.), or to improve services (demand-based transport planning, etc.).

Instead, the idea of *privacy-preserving data analysis* has emerged in recent years: Here, the data is no longer stored in plaintext by the operator, but is protected in some way. One popular approach to obtain aggregate statistics is to not store individual data points but only update aggregates. This works well in practice if only aggregate statistics are needed, but does not work for general analysis, such as usage-based fees or targeted ads. Another approach is to apply differential privacy mechanisms during data collection, so the data being stored and processed is no longer personal data. But this has the drawback that a large amount of noise must be added during data collection to achieve a meaningful level of privacy, which reduces the utility too much for many use cases.

In contrast, we propose a method that collects precise data but protects the link between the data and the individual it belongs to. We can then perform desired analyses and apply differential privacy mechanisms to the output if needed. To this end, we use *secure multi-party computation (MPC)* as a central component. MPC is a cryptographic building block that allows multiple parties to evaluate a common function without learning the input (and output) of the other parties. Thus, with MPC, multiple operators, each owning a part of the data, can perform joint computations on the union of their data sets.

The use of MPC for privacy-preserving data analysis has been studied extensively, see for example [LP09]. However, in typical applications of MPC, the data owners are all directly involved in the computation, e.g., banks or hospitals wishing to obtain results based on their combined customer data, but each having full access to their own customer data. In contrast, we consider a setting *where no single operator has access to the user data to be analyzed.*

However, such an integrated framework *combining privacy-preserving data collection, storage, and analysis* has received little attention in the literature. Currently, there is only one *generic* system for privacy-preserving bookkeeping and analysis [Fet+22b] and few proposals for specific application scenarios (see Section 4.1.3).

For designing a bookkeeping and analytics framework, there are two architectural options regarding data storage: either on the *user's side*, i.e., each user keeps track of its own collected data, or on the *operator's side*. We now take a closer look at both options.

If data is stored on the user's side until analysis, the user has full control over their own data at all times. This is very privacy-preserving, but it also has drawbacks: Users must

[2] See https://www.csoonline.com/article/534628/the-biggest-data-breaches-of-the-21st-century.html and https://www.itgovernance.co.uk/blog/list-of-data-breaches-and-cyber-attacks-in-2023 for examples.

explicitly cooperate and provide their data to the MPC protocol each time data analyses are to be performed. It is highly impractical to require active user participation for recurring automated tasks such as usage-based invoice generation or to obtain usage statistics. Also, users are responsible for the availability of their data, i.e., a lost smartphone on which the collected data was stored might result in the need to pay the maximum amount in a usage-based fee model. Furthermore, mechanisms to ensure the authenticity and "freshness" of the collected data during analysis are required to prevent users from selectively omitting some data points when providing their collected data. This results in the need for heavier crypto components on the user's side, which impact performance for users who may only have a resource-constrained device, and can make the protocol infeasible in practice.

When data is stored on the operator's side, users must give up control over their data, but are no longer responsible for securing their data. However, it is now imperative to ensure that the data is stored securely and cannot be extracted in plaintext by an operator. This can be achieved for example through secret-sharing the data between the operators and performing the analysis with MPC, or by using fully homomorphic encryption (FHE). Since users do not have control over how their data is used in this approach, it is important that the protocol allows only functions approved by the consortium of operators to be evaluated on the data, and that this consortium has enough members who will object to the evaluation of analyses that violate user privacy (see Section 4.2.1 for further discussion).

4.1.1. Contribution

We introduce **POBA**, a generic building block for **P**rivacy-preserving **O**perator-side **B**ookkeeping and **A**nalytics. POBA consists of a consortium of operators and many users, where users collect data to store in *logbooks*, and operators can collectively evaluate analysis functions involving the logbooks of (subsets of) all users. POBA can be used for many different applications that want to combine privacy-preserving data collection, storage, and analysis. POBA is modeled and proven secure in the Universal Composability (UC) framework [Can00; Can20], which ensures that it can be securely integrated into arbitrary applications.

This generic building block has various applications in areas where users need to perform transactions anonymously and without being linked, while still allowing operators to analyze the collected transaction data in the background. Examples of applications include anonymous check-in/check-out multimodal transportation systems, charging/discharging electric vehicle batteries in smart grids, coalition loyalty programs, behavior-based car insurance, and more. Our running example to better explain our system will be a check-in/check-out based payment system for multimodal public transportation: Users can anonymously check in and out of various modes of transportation. Both actions create corresponding log entries, and operators cannot link multiple entries by the same user. At the end of each billing period, an invoice is generated for each user based on the collected log entries. This allows for a sophisticated pricing system that, e.g., combines multiple individual trips into a daily flat rate. Route optimization can be achieved by analyzing trip data from users of multiple transportation companies together. Such systems are already

employed in a non-privacy-preserving matter,[3] where check-ins/check-outs contain a unique identifier of the traveler, allowing for the generation of comprehensive movement profiles. In Section 4.4 we take a closer look at how POBA can be used for anonymous check-in/check-out multimodal transportation systems.

High-Level Idea of our Approach Our system consists of three major steps: data collection, data storage and data analysis.

Data collection: This is done by an (anonymous) user agreeing with one of the operators what data entry should be added to their logbook. In this step, the content of the entry is visible to both the user and the operator, but the identity of the user is encrypted. The fact that user and operator agree on the content of each entry prevents malicious users from sabotaging the analysis by submitting invalid or malformed data.

Data storage: This is handled by secret-sharing the entry between the operators and securely recovering the identity of the user through MPC to obtain the correct logbook to insert it. To prevent the user identity from being leaked in this step, oblivious RAM (ORAM) is used to protect the logbook address, and thus the user identity associated with it. We choose to store data on the operator's side to avoid the need for active user interaction for analysis evaluation and the problem of users losing data, and to reduce the workload on user devices.

Data analysis: Finally, data analysis is performed by evaluating functions approved by the consortium of operators through MPC on (a subset of) the user logbooks. As the logbooks are secret-shared between the operators, no single operator (or collusion of a minority of operators) can directly access the data or evaluate unapproved functions.

Having the operator agree on the content of each entry is important to ensure correctness and authenticity of the collected data. While in some use cases it would be possible for users to prove that their entry is well-formed, it is generally not possible to ensure the authenticity of the data in this way. This means that our system is not suitable for applications in which the content of an entry alone (without the connection to the user's identity) is sensitive, e.g., because the user's identity can directly be inferred from it. However, our system is well suited if the sensitive information is the link between data and user identity, particularly when the entry contains data inevitably observable by the operator (e.g., the presence of some user/vehicle at a specific position, the items on a shopping receipt). A consequence of this is the importance of preventing linkage between the creation of an entry and its usage during analytics, which might involve only a single user's logbook. We make use of oblivious RAM to break this link, ensuring that the information which memory was accessed during a particular analytics evaluation provides no information about when the entries in the involved logbooks were created, without needing to access *all memory* to do so. Otherwise, an operator could for example track

[3] E.g., in transport for NSW https://transportnsw.info/tickets-opal/opal/contactless-payments or transport for London https://tfl.gov.uk/fares/how-to-pay-and-where-to-buy-tickets-and-oyster/pay-as-you-go

the memory address accessed when storing each check-in entry and then later correlate the addresses accessed to create the invoice of a user with this list. We note, however, that even when hiding the content of each entry from the operator during creation, this linkage still needs to be prevented, as it would for example still allow the creation of pseudonymous movement profiles from the interactions with stationary terminals.

Formal Security Model We formalize security (and functionality) for POBA through an ideal functionality $\mathcal{F}_{\text{POBA}}$ modeled in the UC framework. This functionality captures the properties we want a POBA system to have in an ideal world: Users can create an account with one of the system operators, who becomes that user's *home operator*. Users can also create logbook entries with any operator (not just their home operator). The functionality ensures that only registered users can do this, that users can only add entries to their own logbook, and that the user and the operator agree on the content of the new logbook entry. The created logbook entries are then added to the central data storage, where they can only be accessed again when all operators agree to perform a data analysis. This prevents accidental data leaks during storage. Logbook entries are also anonymous and unlinkable, i.e., the identity of the user remains hidden from the operator during entry creation, and no operator can link different entries from the same user. The functionality also enforces that analyses can only be performed if all operators participate and agree on the analysis function. It further guarantees that the analysis function does not reveal additional private data beyond the intended output.

Provably secure protocol We present a protocol Π_{POBA} that UC-realizes the functionality $\mathcal{F}_{\text{POBA}}$ in the presence of a static adversary that may corrupt any number of users and up to $t \leq \frac{n-1}{2}$ of the n system operators, where corrupted users may behave maliciously, while corrupted system operators are assumed to behave semi-honestly. It uses cryptographic building blocks such as threshold signatures, threshold encryption, non-interactive zero-knowledge (NIZK) proofs, secret-sharing, and MPC that supports ORAM. One of the challenges has been to carefully combine the different primitives and techniques into a UC-secure protocol, while also providing practical efficiency. To this end, it is important to only prove statements in zero-knowledge for which efficient proofs are possible, to take care of what needs to be evaluated inside MPC, and to consider the limited computational power of users. In particular, we assume that users run constrained devices such as smartphones or smartwatches, and minimize the computation and communication required from them.

Evaluation and Benchmarks We implemented Π_{POBA} to evaluate its practicality and to determine which cryptographic building blocks are the bottlenecks. Our results show that the performance of ORAM within secure computation is currently a significant bottleneck, limiting practicality to a small number of operators and users, especially since this step does not parallelize across more hardware. All other parts of our protocol are efficient enough for practical use, and the operator computation besides ORAM accesses can be parallelized across additional hardware to scale with the number of users. There have

been many efforts in recent years to develop protocols for secure computation on ORAM for the special cases of $n = 3$ (or 4) and $t = 1$, but none of these generalize to a setting with $t > 1$ as n increases. For a setting with $t \leq \frac{n-1}{2}$ corruptions, the state of the art is still a variant of PathORAM [Ste+13]. Possible future optimizations in this setting will directly translate to a respective performance increase of our protocol.

Using the MP-SPDZ framework [Kel20] with its implementation of PathORAM [KS14] for $n = 9$, $t = 4$, up to 2^{15} registered users and logbooks with up to 100 entries, our protocol has a throughput of ≈30k entries per day. Regarding our transportation payment example, this would translate to a small city of 15k people with an average of one trip per person per day and a limit of 50 trips per person per week when one week is used as the period duration.

In the restricted $n = 3$, $t = 1$ setting, GigaDORAM [Fal+23] reports 700 queries/s even with memories of size 2^{30}, indicating an access time of less than 2 ms. Using GigaDORAM, we achieve a throughput of over two million inserts per day with one million registered users, corresponding to a medium-to-large European city.

Overall, our benchmarks already suggest that when restricting to a small number of operators and moderate number of users, POBA could be practical in a check-in/check-out transportation scenario for a city up to a few hundred thousand people. Since our protocol uses MPC with ORAM as a black-box building block, any future improvements there can be directly applied to our protocol.

Our implementation is available at `https://github.com/kastel-security/poba`.

4.1.2. Overview of Technical Challenges

Designing and implementing a cryptographic protocol with the complexity of POBA comes with a number of challenges. First, secure computation protocols are comparatively slow, and making users wait for or even participate in them reduces user satisfaction or might simply be impossible. But in many use cases, it is sufficient for user and operator to agree on a new log entry, and the actual writing to the logbook can be delayed until a later time. This is especially useful in situations where timely feedback to the user is important, but the operator's device used for the interaction has spotty connectivity, which may be the case for check-ins/check-outs in mobile vehicles such as trains. Therefore, we split the writing of new entries into two tasks: In the first task, the user and an operator agree on the new entry, and in the second task, the operator interacts with the other operators to actually write the entry into the correct logbook. In this scenario, the user is no longer needed for the second task.

Second, when implementing an oblivious logbook, letting its size grow arbitrarily leads to (some information about) the current size being leaked when accessing it. In addition, large logbooks have a large performance impact. To deal with this problem, we divide logbooks into periods, with a fixed size for each period. When creating new entries, user and operator now also agree on the period in which the entry should land. This allows

new entries to be inserted efficiently, and analysis calculations can specify the relevant periods and thus benefit from better efficiency when only a subset of periods is needed.

Third, it may be necessary for a user to convince an operator that an entry was made previously. Again, proof of check-in is important for spot checks when considering check-in/check-out-based public transportation payment systems.

Fourth, some way of mapping users to logbooks (i.e., locations in memory) is required. This mapping must be done in a way that prevents impersonation of honest users, even by an adversary corrupting several operators. An intuitive solution would be to use a user's public key as the address of the logbook, and to require a proof-of-knowledge of the secret key when creating new entries. But for this to be secure, the public key has to be large, i.e., λ bit, or $\geq$250 bit when using elliptic curves, while the number of users is typically much smaller, i.e., 2^{20} to 2^{30}. It is not feasible to allocate memory for 2^{λ} potential logbooks. Hashing the public key into a smaller address requires a collision-free hash function, since collisions would allow impersonation of honest users. But using a collision-resistant hash function still results in addresses of λ bit length. Another method is to create a fixed mapping from a user's public key to their logbook address. This mapping could then be stored in a hash table, where collision-freeness can be guaranteed by choosing a new random universal hash function whenever inserting a new entry would result in a collision. But this approach requires that the hash table is queried in an oblivious way, and thus requires another layer of oblivious RAM. Since ORAM is the bottleneck of our protocol, we decided against this approach and instead chose the following solution: The mapping is given to the user, along with a signature on the public key/address pair. The user then provides a zero-knowledge proof of knowledge of the secret key along with the signature on the public key/address pair, and an encryption of the address. This introduces another challenge: Even semi-honest corruption of an operator leaks their secret key to the adversary. This secret key can then be abused by malicious users to generate signatures that map their public key to the address of an honest user's logbook. To prevent this, we use a threshold signature scheme, which requires secret keys of at least $t + 1$ operators to generate a valid signature.

The next challenge is to encrypt and sign the address in a way that is compatible with both efficient zero-knowledge proofs and decryption within a secure computation protocol. Groth–Sahai proofs are very efficient when working with structure-preserving primitives. Thus, using a structure-preserving threshold signature scheme and encoding the address as a point on the elliptic curve is the natural answer. The latter is typically done by multiplying it by for example 2^{10}, interpreting this as the x-coordinate, and then adding 1 to this value until one ends up with a valid point on the curve. But this only works if the subgroup used has a small cofactor. The popular BLS curves use subgroups with very large cofactors, on the order of 2^{100}, which results in having to do around 2^{100} iterations before finding a valid point in the subgroup. Fortunately, Barreto–Naehrig curves, which have fallen out of favor due to new attacks that mandated larger parameters to maintain 128 bit security, use the whole curve as $\mathbb{G}_1$ resulting in cofactor 1 and thus allow this type of encoding.

Fifth, secure computation works over a finite field or ring, the size of which has a significant impact on performance. To allow decryption of a ciphertext over a Barreto–Naehrig curve with ≈128 bit security, it is necessary to work over a finite field of size $\approx 2^{382}$. Performing oblivious RAM operations on this field has a prohibitive performance cost. As a solution, we split our secure computation into two parts: First, the ciphertext is decrypted over the large field. Then, using the share conversion method of [DT08], the resulting plaintext (which is known to be an address much smaller than 382 bit) is converted to a smaller field, and the secure computation involving oblivious RAM is performed over this smaller field. Specifically, we operate on a 128 bit field for this.

4.1.3. Related Work

Privacy-preserving data analysis is not only a topic discussed in society, but it will also become an important economic factor in the future: The global market size for privacy-enhancing computation was already estimated at $2.2 billion for 2023 and is projected to reach approximately $24.7 billion by 2033.[4] As a consequence, we should make sure that there are cryptographically sound methods to protect our personal data during the full data analysis life cycle. For the setting where a small number of data-holders want to evaluate functions on their combined data, i.e., banks performing clustering on their combined transaction graphs for money-laundering detection, standard MPC approaches can easily be employed. However, when involving the data of a large number of users without huge computational resources, mechanisms to collect the data in a privacy-preserving manner have to be combined with mechanisms to securely evaluate analysis functions.

When reviewing the prior work on the combination of privacy-preserving data collection, storage and analysis, we distinguish between solutions targeting specific applications and generic solutions.

Generic solutions for user data collection and analysis mainly fall into two categories: applying noise to the data during collection to achieve differential privacy, or being limited to aggregation functions (e.g., vector sum, heavy hitters) over the collected data.

Differential Privacy during Data Collection There are multiple systems employing differential privacy during data collection to protect user privacy. Examples include RAPPOR [EPK14; FPE16], which is used in Chromium. The downside of this approach is that when adding little noise, the system provides weak privacy guarantees, whereas when adding a lot of noise, the system provides low utility.

Privacy-Preserving Aggregate Statistics Several protocols to generate aggregate statistics in a privacy-preserving way have been proposed [MDD16; CB17; Bon+21; Bel+23; Add+22; MST24]. These protocols allow computation of aggregation functions, such as

[4] https://www.precedenceresearch.com/privacy-enhancing-computation-market

the sum of input vectors (e.g., Prio, Prio+ [CB17; Add+22]) or heavy hitters and subset-histograms (e.g., Poplar, Doplar [Bon+21; Dav+23]). To this end, one input per client is aggregated over an epoch and then the output of the function is revealed. The drawback of these approaches is that the class of analysis functions that can be evaluated is constrained, in particular no analyses regarding multiple inputs (from different points in time) of one user can be obtained, as would be required for scenarios such as usage-based fee calculation or targeted advertisements.

Generic Privacy-Preserving Data Collection and Analytics The only solution for privacy-preserving data collection and analytics that allows generic analysis functions we are aware of is PUBA [Fet+22b], which also comes with a formal security model and proof in the UC framework. In PUBA, logbooks, which are called "user histories" there, are stored on each individual user's device. Simple single-user analysis functions can be evaluated by the user itself and correctness of the result proven in zero-knowledge, similar to the smart-metering setting of [RD11; RDK18], but since user devices are assumed to be smartphones the class of functions is even more limited. For more complex functions or functions that rely on data from multiple users, PUBA has each user secret-share their logbook between the system operator and a non-colluding "proxy" party, who then evaluate the function using MPC.

While our setting is similar to theirs, we address several shortcomings: Firstly, we allow multiple operators to collectively operate the system (PUBA does not generalize to multiple distrusting operators, as each operator could arbitrarily modify the content of user logbooks), removing the need for an external party doing heavy computations. Second, we store logbooks securely between the operators instead of on the user's device, providing robustness against data loss from lost/stolen/broken user devices and reducing the computation and communication required from users. Third, *analysis computations no longer require user interaction*, making a deployment much more practical. This reduces the delay between start and completion thereof and ensures that data from the involved users covers the same time period (in PUBA, this task needs to wait for input from each user, and users input their current logbook, causing users that provide input at a later point in time to cover a later time period). Fourth, with our concept of time periods, we support logbooks that grow over time, whereas PUBA has one fixed logbook size and old entries get overwritten over time.

Privacy-Preserving Ticketing for Public Transportation Several proposals exist for privacy-preserving ticketing systems in public transportation [Mil+15; Hey+06; Arf+15; Gud14; Rup+13; KLG13; Viv+10; JJQ07; Han+21]. However, most of these systems focus solely on payment functions and do not address the privacy-preserving analysis of resulting data. Kerschbaum et al. [KLG13] appear to be an exception, as they are developing a privacy-preserving ticketing system for public transportation that stores the (encrypted) data in the backend. The travel data is encrypted using an additively homomorphic encryption scheme, where the decryption key is held by a trusted third party, who needs to be involved in entry operations, invoice calculation, and any analysis function (modeled as a circuit)

that requires negation. Unlike POBA, their system only contains a single provider and they have no formal model or even UC proof. Furthermore, of these systems only [Han+21] has a formal security proof, however only in a standalone simulation-based model, leaving security under arbitrary composition open.

Privacy-Preserving Electric Vehicle Charging/Discharging To enable privacy-preserving billing in smart-grid scenarios where electric vehicles are charged/discharged based on the grid state, several works [Au+14; Zha+14; Sch+19; Wan+22; Yue+24; Che+24] propose solutions for billing. Of these, [Sch+19; Wan+22] have proofs in the UC framework. However, they all only compute billing information and do not allow any further analytics.

4.2. An Idealized POBA model

In this section, we provide an overview of our system and the parties involved. Figure 4.1 gives an overview of how the involved parties could interact with our system in the context of an anonymous check-in/check-out based multimodal transportation system.

4.2.1. Parties and Trust Assumptions

Users (U) interact with the system by collecting data in their individual logbooks. Our system allows for an arbitrary number of users who are untrusted (malicious) and may collude with other corrupted parties. Users are assumed to use comparatively weaker hardware, and thus their computational burden is kept as low as possible.

System operators (O) run the system. They usually provide some kind of service that users are interested in, and as part of that service they want to collect data from users. Each user is registered with one system operator, which is called the user's *home operator (HO)*. Note that the set of operators can also be heterogeneous, since different kind of operators may have different interests in the collected data. For example, in the charging and discharging electric vehicle batteries in the smart grid scenario, operators could not only be electricity providers but also car manufacturers who may want to analyze the lifetime of batteries in their vehicles.

Since one of our goals is to evaluate the practicality of POBA, we assume the most favorable corruption model: An adversary may only corrupt a minority of operators, and corrupted operators are semi-honest rather than malicious.[5] While this is a rather strong assumption, we argue that this assumption is appropriate in many cases: It protects against accidental data leaks, curious or disgruntled employees, and data breaches. Cybercriminals are increasingly threatening to release private customer data unless a ransom is paid, as was for example the case with a Finnish mental health care provider,[6] and not having such data in the first place prevents this. Similarly, it is not uncommon for curious employees

[5] While malicious parties can arbitrarily deviate from the protocol, semi-honest parties are assumed to honestly follow the protocol. The goal of a semi-honest party is to extract more information than intended from the protocol. Several semi-honest parties can also collaborate and try to gain additional knowledge from their combined sets of information.

[6] https://www.wired.com/story/hacker-threaten-release-therapy-notes-patients/

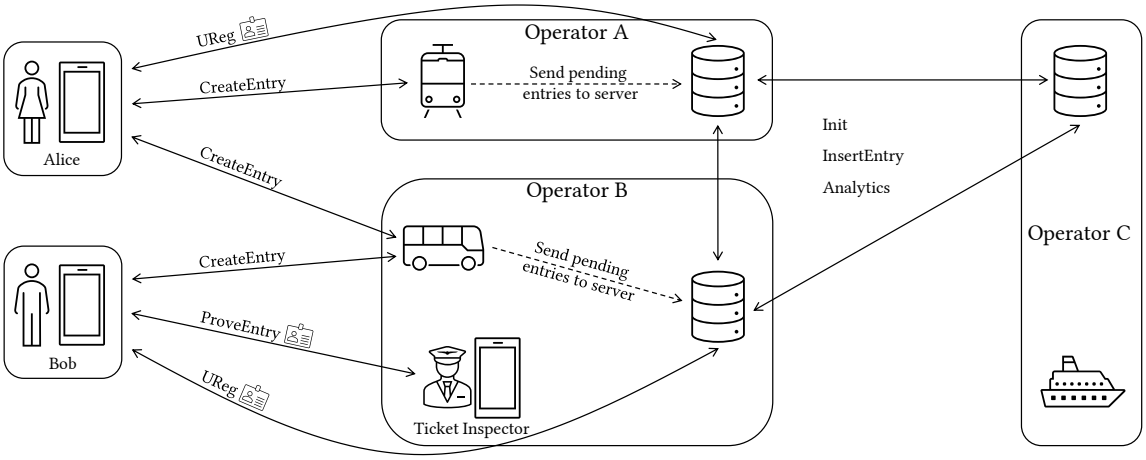

Figure 4.1.: Intended use of the system using the application example check-in/check-out transportation systems with different mobility providers. Arrows with a 🪪 represent identifying interactions and dotted arrows are assumed to happen outside of the protocol.

to violate the privacy of customers.[7] Active manipulation of installed software is a much higher barrier than abusing legitimate access mechanisms, and tampered software could also be detected by remote attestation mechanisms.

When performing analytics, we assume that operators agree among themselves beforehand which analytics they will run on which data. Since the majority of operators are assumed to be honest, they will not agree to perform analyses that violate privacy rules. If desired, privacy advocates could be allowed to participate in the MPC as observers to ensure that the analysis function protects the privacy of user data.

4.2.2. Logbooks and User Data

Logbooks are at the heart of our system: All the data collected by a user is stored in them. As mentioned above, there is a separate logbook for each user. Each logbook consists of a list of entries, where each entry has a fixed (maximum) length. For efficiency purposes, logbooks are further divided into time periods and have a maximum size (i.e., number of entries). The length of a period and the size of a logbook should be chosen so that the logbooks don't overflow. The logbook for user U and period p is referred to as Log_U^p. When operators want to perform analysis, these logbooks are used as a data basis.

To prevent malicious users from sabotaging analyses by submitting invalid or malformed data, we require the user to agree with one of the operators on the content of each entry. Thus, the system is well suited for purposes where the data in each entry itself is not personal information, but the combination of data and user identity is relevant and sensitive.

[7] https://interestingengineering.com/culture/google-fires-80-employees-for-exploiting-user-data

4.2.3. Desired Security Properties

Informally, we want to achieve the following guarantees:

- Users can only create logbook entries for their own logbook, and only after creating an account
- Logbook entries can only be generated if both user and operator agree on their content
- Operators can not identify users during entry creation
- Operators can not link different entries to the same user
- Users can prove that they created a specific logbook entry if and only if they actually did so
- Computation of an analytic function does not reveal additional private data besides the intended output
- Computation of an analytic function returns the correct output

It is easy to see that our ideal functionality satisfies these.

4.2.4. Our Ideal Functionality

$\mathcal{F}_{\text{POBA}}$ internally manages all logbooks and pending entries, and provides system operators with the output of analytic functions they all agree to evaluate. In Fig. 4.1 we illustrate the interplay of the different tasks using the check-in/check-out based transportation system of Section 4.4.1 as an example. We now provide both a high-level description of these tasks and a formal description in Fig. 4.2.

Notation Our ideal functionality is described as a set of tasks that can be called by the parties. Each task has inputs of the form "Input P: (`cmd`, *ssid*, *msg*)" (where P is the party that should provide this input, `cmd` specifies the task, *ssid* is the subsession id, and *msg* is the input of that party for the given task), a behavior, and outputs of the form "Output P: (*ssid*, *msg*)" (where *ssid* is the same subsession id and *msg* the output for that party). This means that after the functionality receives messages from all parties P specified as having input with the same *ssid*, it runs the instructions listed under "Behavior". If doing so does not cause it to ignore the inputs, it makes private delayed outputs to the parties P with content (*ssid*, *msg*) as listed under "Output". If the party for which output is generated is semi-honest, the adversary is additionally provided with (*ssid*, *msg*). Additionally, whenever a semi-honest party P provides input (`cmd`, *ssid*, *msg*), it sends (`cmd`, *ssid*, *msg*, P) to the adversary. When an honest operator O provides input, it sends (`cmd`, *ssid*, O) to the adversary. When an honest user U provides input, if `cmd` = `ureg` it sends (`ureg`, *ssid*, U) to the adversary, for `cmd` $\neq$ `ureg` it instead sends (`cmd`, *ssid*, `user`) to the adversary, i.e., it keeps the identity of the user secret.

Functionality $\mathcal{F}_{\text{POBA}}$

State:
- Set of all operators $\mathcal{O}$, logbook size *size*
- Counter id^{O}_{last} for each $O \in \mathcal{O}$ to enumerate pending log entries
- List $\mathsf{L}_{\text{users}}$ for each $O \in \mathcal{O}$ containing a list of users registered with that operator
- List $\mathsf{L}^{\text{O}}_{\text{Pending}}$ for each $O \in \mathcal{O}$ to store pending log entries
- Logbooks Log^{p}_{U} for each user U and period p
- Variable *initialized*, initially set to 0

Init:
- Input each O: (`init`)
- Behavior:
  1. After receiving input from all $O \in \mathcal{O}$, set *initialized* $\coloneqq 1$
- Output each O: (`init`, ok)

Ignore messages other than `init` until *initialized* = 1.

UReg:
- Input U: (`ureg`, *ssid*, pid_{HO})
- Input HO: (`ureg`, *ssid*, pid_U)
- Input all other O': (`ureg`, *ssid*)
- Behavior:
  1. If U is already in $\bigcup_{O \in \mathcal{O}} \mathsf{L}_{\text{users}}$ or U does not have pid pid_U or HO does not have pid pid_{HO}, ignore this call
  2. Upon receiving input from HO, send ((`ureg`, *ssid*, pid_U), HO) to the adversary
  3. Add (pid_U) to $\mathsf{L}^{\text{HO}}_{\text{users}}$
- Output U: (*ssid*, ok)
- Output HO: (*ssid*, ok)

CreateEntry:
- Input U: (`createentry`, *ssid*, $period_U$, $entry_U$, pid_O)
- Input O: (`createentry`, *ssid*, $period_O$, $entry_O$)
- Behavior:
  1. If pid_U is not in $\bigcup_{O \in \mathcal{O}} \mathsf{L}_{\text{users}}$ or $entry_U \neq entry_O$ or $period_U \neq period_O$ or O does not have pid pid_O, ignore this call
  2. Set $id^{O}_{\text{last}} \coloneqq id^{O}_{\text{last}} + 1$, $id \coloneqq id^{O}_{\text{last}}$
  3. Add (id, pid_U, *period*, $entry_O$) to $\mathsf{L}^{\text{O}}_{\text{Pending}}$
- Output U: (*ssid*, ok)
- Output O: (*ssid*, ok, *id*)

InsertEntry:
- Input O: (`insertentry`, *ssid*, *period*, *id*)
- Input all other O': (`insertentry`, *ssid*)
- Behavior:
  1. If no entry (id, pid_U, *period*, *entry*) exists in $\mathsf{L}^{\text{O}}_{\text{Pending}}$, ignore this call
  2. If at least one O' is corrupted and O is honest, leak ((`insertentry`, *ssid*, *period*, *entry*), O) to the adversary
  3. If $|\text{Log}^{period}_{U}| = size$, output (*ssid*, `error`) to all O
  4. Otherwise:
     a) Remove that entry from $\mathsf{L}^{\text{O}}_{\text{Pending}}$
     b) Append (*entry*) to Log^{period}_{U}
- Output all O: (*ssid*, ok)

ProveEntry:
- Input U: (`proveentry`, *ssid*, *entry*, *period*, pid_O)
- Behavior:
  1. If no entry (id, pid_U, *period*, *entry*) in $\mathsf{L}_{\text{Pending}}{}^{O'}_{O' \in \mathcal{O}}$ or (*entry*, O') in Log^{period}_{U} exists, ignore this input
  2. If both U and O' such that the entry was in $\mathsf{L}_{\text{Pending}}{}^{O'}$ resp. the entry (*entry*, O') existed in Log^{period}_{U} are honest, send (`proveentry`, *ssid*, leak, O') to the adversary
- Output O: (*ssid*, `proveentry`, pid_U, *entry*, *period*)

Analytics:
- Input all O: (`analytics`, *ssid*, *func*, $\{period_i\}$, $\{U_i\}$, aux_O)
- Behavior:
  1. If not all parties provided the same *func*, $\{period_i\}$ and $\{U_i\}$ as input, ignore this call
  2. Set Logs $\coloneqq \{\text{Log}^{p}_{U}\}_{U \in \{U_i\}, p \in \{period_i\}}$
  3. Evaluate $\{\text{result}_O\}_O \leftarrow func(\text{Logs}, \{aux_O\}_O)$, setting *sequence* $\coloneqq (p_1, p_2, \ldots, p_n)$ with an entry p_j for each time *func* reads an entry from $\text{Log}^{p_j}_{U}$
  4. If at least one operator is corrupted, leak (`analytics`, *ssid*, *sequence*) to the adversary
- Output each O: (*ssid*, result_O)

Figure 4.2.: Ideal functionality $\mathcal{F}_{\text{POBA}}$

Init To model required setup steps in the real protocol, the functionality only starts working after each operator called this task.

UReg Before participating in the system, a user has to select a home operator HO from the list of operators and register with this operator. This ensures that information about user accounts, which may also contain (static) personal data (e.g., billing information), is only held by one operator. All other operators need to acknowledge the new user. The functionality then adds the user to the list of registered users with that home operator.

CreateEntry This task initiates creation of a new logbook entry. First, user and operator agree (out-of-protocol) on the content *entry* of the new entry and which period p it belongs to. Then, the functionality ensures that the user has previously registered an account,

while keeping the user's identity secret. The entry is then added to a list of pending entries that have to be inserted into the correct logbook using the next task.

InsertEntry In this task, all operators work together to move an entry from the pending list into the correct logbook. Only after an entry has been inserted by this task can it be included in analytics computations.

ProveEntry As mentioned above, in some applications it is useful for users to be able to prove that their logbook contains certain (pending) entries, e.g., spot checks can be implemented by users showing that they created a check-in entry for the current train. This task does just that: A user can prove to an operator that they produced a certain logbook entry *entry* in period p. To do this, the functionality sends the identity of the user together with the period and entry to the operator if the entry is stored in either the user's logbook or the list of pending entries.

Analytics In this task, all operators work together to evaluate an analysis function that they have previously agreed on out-of-band. The analysis function can use the logbooks of several users from several periods as a data basis. The functionality first ensures that all operators agree on what function to evaluate on which users and periods. Each operator receives an output from the analysis, but it is possible that each operator receives a different output (depending on the specific analysis function). It depends on the application what exactly the analysis function calculates. For example, it can be used to create customer invoices and conduct market research, compute statistics to improve a service, etc.

4.3. A Protocol to Realize the Model

In this section we present our protocol Π_{POBA} that UC-realizes the ideal functionality $\mathcal{F}_{\text{POBA}}$.

At the heart of our system is the oblivious logbook storage, which is built upon oblivious RAM (ORAM), or more precisely an MPC protocol with ORAM. For each user, memory in the ORAM is reserved and logbook entries are written there. To allow for efficient user-facing protocols, logbook entries are first stored locally at the respective operator with an encrypted user identifier and later written into the actual logbook in batches. For better efficiency, we also split logbooks into periods, where each period starts with a new empty logbook. This is realized by simply having different instances of ORAM for each period, where a new instance is initialized as empty the first time an entry is generated for it. Note that analytics can still be based on multiple periods if desired.

4.3.1. Building Blocks

Before delving into the actual protocol, we briefly introduce the building blocks we use to achieve the desired security guarantees on a high level. Formal definitions are given in Chapter 2.

Threshold Public Key Encryption (PKE) We use a threshold PKE to encrypt user identities. A threshold PKE with n parties and threshold t has a key generation algorithm that outputs an encryption key ek along with n decryption key shares dk_i. It has a standard encryption algorithm, but for decryption it has a partial decryption algorithm that, given a ciphertext and a decryption key share, outputs a decryption share, and a combine algorithm that combines $t + 1$ of those decryption shares to recover the plaintext. We use this to encrypt user identities when creating new logbook entries to prevent a colluding minority of operators from decrypting those identities. IND-CPA-security guarantees that, given the encryption key and at most t decryption key shares, it is hard to decide whether a given ciphertext is an encryption of m_0 or m_1 for any same-length messages m_0, m_1.

See Definition 2.7.3 for a formal definition. Additionally, we require a distributed key generation protocol, formalized as $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$, see Fig. 2.3a.

Digital Signatures A digital signature scheme consists of a key generation algorithm that outputs a verification key vk and a signing key sk, and algorithms for signing messages with the signing key and verifying signatures with the verification key. EUF-CMA-security guarantees that it is hard to generate valid signatures given only the verification key. We use a (regular) signature scheme to authenticate log entries.

In aggregate signature schemes it is possible to combine different signatures into one aggregate signature, which reduces signature size and speeds up verification. We use this to efficiently verify system keys.

In a threshold signature scheme multiple parties have to work together to create a signature on a message. We use this to authenticate user credentials. It prevents a single leaked operator signing key from creating fake credentials for honest users, which would allow an adversary to forge entries for that user's logbook. The threshold version again has a key generation algorithm that outputs a verification key vk and signing key shares sk_i. Signing messages consists of generating a partial signature with a signing key share and then combining $t + 1$ such partial signatures into a full signature. Here we use the TS-UF-1-security definition, which guarantees that given at most t signing key shares, the verification key, and access to an oracle that generates partial signatures, it is hard to generate a valid signature on a message that has not been queried to the oracle.

Formal definitions are given in Definition 2.8.1 for digital signatures, in Definition 2.8.2 for EUF-CMA-security, in Definition 2.8.4 for aggregate signatures, in Definition 2.8.5 for threshold signatures, and in Definition 2.8.6 for TS-UF-1-security. Again, we additionally require a distributed key generation protocol, formalized as $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$, see Fig. 2.3b.

Non-Interactive Zero-Knowledge Proofs To allow user anonymity, we use non-interactive zero-knowledge proofs of knowledge in the common reference string (*crs*) model. Such a proof allows a party to convince another party that a statement is in some NP relation, and even that they know a witness for this fact, without revealing information about the witness. More precisely, given a relation R, a statement *stmt*, and a witness *wit*, such that $(stmt,\ wit) \in R$, there is a prove algorithm for generating a proof and a verification algorithm that, given the relation R, the statement *stmt*, and the proof, verifies that it is valid. Additionally, given some trapdoor information that could be generated during setup, there exist algorithms to simulate proofs without actually knowing a witness, and to extract the witness from proofs. The zero-knowledge property guarantees that simulated proofs are indistinguishable from real proofs and *simulation-extractability* guarantees that extraction with the trapdoor is possible even if the adversary has seen simulated proofs before.
For formal definitions, see Definitions 2.5.1 and 2.5.4.

Secure Computation of RAM Programs We use MPC to manage the logbooks and compute analytics. To make this feasible with a large number of users, we need a secure computation protocol that allows the evaluation of RAM programs: Usually, secure computation is defined in terms of circuits, which has the advantage of being side-channel free. However, in order to access an input-dependent entry of an array, input wires for all entries of that array must go into a large multiplexer circuit that selects the correct entry. In our case, the array corresponds to the collection of logbooks for all users, which can range from tens of thousands to several million, so this is clearly not practical. When using a RAM program, on the other hand, accessing such an entry simply corresponds to a single memory access at the desired location, the complexity of which is independent of the size of the array. However, doing so naively reveals *which* entry of the array was accessed. So we require our MPC protocol to access the RAM in an oblivious way. To achieve this, oblivious RAM [Gol87] was introduced. It can be viewed as a compiler that transforms logical memory accesses into a sequence of physical memory accesses that look random. Combined with secure multi-party computation, this allows us to access a single logbook without revealing which one it was (and thus which user it belongs to), with a complexity that grows only logarithmically with the number of users. For modeling this, we follow the approach of [Bra+23] and use an ideal functionality $\mathcal{F}_{\mathrm{MPC}(f)}$ to securely evaluate a RAM program f. To this end, we require a secret-sharing scheme S which consists of algorithms to generate n shares from a secret such that up to t of those shares reveal no information about the secret, but any $t+1$ shares can be used to reconstruct the secret. See Definition 2.6.1 for a formal definition. $\mathcal{F}_{\mathrm{MPC}(f)}$ then takes as input secret-shares of the memory M and from each party P_i some input $\mathtt{in}_i$, reconstructs M from these shares, executes $f((\mathtt{in}_1,\ \ldots,\ \mathtt{in}_n),\ M)$ and outputs the result together with the number of memory accesses performed by f, as well as new shares of the updated memory. In addition, we use an extended functionality $\mathcal{F}^{\mathrm{ext}}_{\mathrm{MPC}(f')}$ that takes a collection of memories $M_1, \ldots, M_n$ as input and an extended RAM program f' that can perform memory accesses on any of these memories by specifying the index of the memory. Given a protocol realizing $\mathcal{F}_{\mathrm{MPC}(f)}$, it is straightforward to obtain a protocol realizing $\mathcal{F}^{\mathrm{ext}}_{\mathrm{MPC}(f)}$ by

Functionality $\mathcal{F}_{\mathbf{MPC}(f)}$

Parameters: A next-step function NS, number of participating parties n

- Upon receiving (Run, $\langle M \rangle$, $input_i$) from all parties P_i, $i < n$:
  1. M = S.Recon($\langle M \rangle$)
  2. Initialize $\text{st}_{\text{NS}} \coloneqq (\text{start}, (input_1, \ldots, input_n))$, $data \coloneqq 0$ and $round \coloneqq 0$
  3. While $\text{st}_{\text{NS}} \neq (\text{stop}, (output_1, \ldots, output_n))$, do:
     a) Run $(\text{st}_{\text{NS}}, I) \leftarrow \text{NS}(\text{st}_{\text{NS}}, data)$
     b) If $I = (\text{read}, addr, \perp)$, set $data \coloneqq M[addr].val$
     c) Else if $I = (\text{write}, addr, val)$, set $data \coloneqq M[addr].val$ and update $M[addr].val = I.val$
     d) Update $round \coloneqq round + 1$
  4. $\langle M^{\text{new}} \rangle \leftarrow$ S.Share(M)
  5. Output $(\langle M \rangle_i, output_i, round)$ to each party P_i, $i < n$.

(a) Functionality $\mathcal{F}_{\text{MPC}(f)}$ for securely evaluating RAM programs.

Functionality $\mathcal{F}^{\text{ext}}_{\mathbf{MPC}(f)}$

Parameters: A next-step function NS, number of participating parties n

- Upon receiving (Run, $\{\langle M_j \rangle\}_{j<k}$, $input_i$) from all parties P_i, $i < n$:
  1. For each $j < k$: M_j = S.Recon($\langle M_j \rangle$)
  2. Initialize $\text{st}_{\text{NS}} \coloneqq (\text{start}, (input_1, \ldots, input_n))$, $data \coloneqq 0$, $round \coloneqq 0$ and $sequence = ()$
  3. While $\text{st}_{\text{NS}} \neq (\text{stop}, (output_1, \ldots, output_n))$, do:
     a) Run $(\text{st}_{\text{NS}}, I) \leftarrow \text{NS}(\text{st}_{\text{NS}}, data)$
     b) If $I = (\text{read}, index, addr, \perp)$:
        i. Ensure $index < k$, otherwise abort
        ii. Set $data \coloneqq M_{index}[addr].val$
     c) Else if $I = (\text{write}, index, addr, val)$
        i. Ensure $index < k$, otherwise abort
        ii. Set $data \coloneqq M_{index}[addr].val$
        iii. Update $M_{index}[addr].val = I.val$
     d) Update $round \coloneqq round + 1$ and $sequence \coloneqq (sequence, index)$
  4. For each $j < k$: $\langle M_j^{\text{new}} \rangle \leftarrow$ S.Share(M_j)
  5. Output $(\{\langle M_j \rangle_i\}_{j<k}, output_i, sequence)$ to each party P_i, $i < n$.

(b) Functionality $\mathcal{F}^{\text{ext}}_{\text{MPC}(f)}$ for securely evaluating extended RAM programs.

Figure 4.3.: Functionalities for secure computation.

simply using multiple independent instances of the mechanism to access the memory. For the formal definitions of these functionalities, see Figs. 4.3a and 4.3b.

(Semi-)Authenticated Secure Communication Our protocol relies on proper authentication during account creation, but also on the ability of users to anonymously communicate with operators during entry creation. We formalize this by using $\mathcal{F}_{\text{SMT}}$ for normal secure authenticated communication and $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$ for secure communication where the initial receiver of the message is authenticated whereas the initial sender remains anonymous. $\mathcal{F}_{\text{SMT}}$ simply forwards a message msg together with the sender pid pid_S to the receiver R with pid pid_R, thus authenticating both the sender and receiver and keeping the content confidential. $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$ on the other hand only forwards msg to the receiver R with pid pid_R but keeps the identity of the sender S secret, thus only authenticating the receiver of the message. To enable communication in both directions, $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$ additionally allows a receiver to respond to the message. For the formal definitions of these functionalities, see Fig. 2.2. Note that we assume all communication between operators to use secure channels and omit the usage of $\mathcal{F}_{\text{SMT}}$ there and will simply write $O \rightarrow O' : (msg)$ as a shorthand for transmitting msg from O to O' using $\mathcal{F}_{\text{SMT}}$.

In practice, this can be achieved through anonymous wireless communication such as NFC or Bluetooth LE or the usage of anonymization networks such as TOR when communication over the internet is required.

Details of the relation $\mathcal{R}_{\mathsf{UR}}$

$stmt \coloneqq (\mathsf{pk}_U)$
$wit \coloneqq (\mathsf{sk}_U)$

$\mathcal{R}_{\mathsf{UR}} = \{(stmt, wit) \mid$
- $\mathsf{KeyVerify}(\mathsf{sk}_U, \mathsf{pk}_U) = 1$

$\}$

(a) Relation $\mathcal{R}_{\mathsf{UR}}$ used during user registration.

Details of the relation $\mathcal{R}_{\mathsf{CE}}$

$stmt \coloneqq (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
$wit \coloneqq (\mathsf{pk}_U, \mathsf{sk}_U, addr, \sigma, r)$

$\mathcal{R}_{\mathsf{CE}} = \{(stmt, wit) \mid$
- $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, addr; r)$
- $\mathsf{TSIG.Vf}(crs_{\mathsf{TSIG}}, \mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) = true$
- $\mathsf{KeyVerify}(\mathsf{sk}_U, \mathsf{pk}_U) = 1$

$\}$

(b) Relation $\mathcal{R}_{\mathsf{CE}}$ used when creating new entries.

Details of the relation $\mathcal{R}_{\mathsf{PE}}$

$stmt \coloneqq (\mathsf{pk}_U, addr, ct_U, \mathsf{ek})$
$wit \coloneqq (r, \mathsf{sk}_U)$

$\mathcal{R}_{\mathsf{PE}} = \{(stmt, wit) \mid$
- $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, addr; r)$
- $\mathsf{KeyVerify}(\mathsf{sk}_U, \mathsf{pk}_U) = 1$

$\}$

(c) Relation $\mathcal{R}_{\mathsf{PE}}$ used when proving an entry was created.

Figure 4.4.: Zero-knowledge relations used in Π_{POBA}.

4.3.2. Protocol Description

We now give a high-level description of our protocol, along with a formal one in Figs. 4.5 to 4.7 and 4.9. The NP-relations used in the NIZK proofs are explained in Fig. 4.4.

Logbooks are identified by their logical memory address, and a list of free addresses $\mathsf{L}_{\mathsf{Free}}$ is synchronized between the operators. When a new user registers, a mapping is created between that user (represented by their public key) and a logbook (represented by its address). The number of logbooks (i.e., both assigned and free addresses) determines the size of the shared memory and the cost of accessing it. Thus, if the ORAM used does not allow the memory size to be increased dynamically, it must be increased for a new period when the number of free addresses becomes too small.

Init To set up the system, each operator generates a signature keypair $(\mathsf{sk}_O, \mathsf{vk}_O)$. They also perform a distributed key generation for the threshold encryption scheme and the threshold signature scheme, captured by $\mathcal{F}_{\mathrm{Keygen}}^{\mathsf{TPKE}}$ and $\mathcal{F}_{\mathrm{Keygen}}^{\mathsf{TSIG}}$ (see Fig. 2.3), to get a common encryption key ek with decryption key shares dk_i as well as a common verification key vk together with signing key shares sk_i.

User Registration To register, a user chooses one of the system operators as their home operator and creates an account with them. The user generates a signature key pair $(\mathsf{sk}_U, \mathsf{pk}_U)$ and proves knowledge of the signing key sk_U in zero-knowledge to the home operator. The home operator then picks a free logbook address *addr* from $\mathsf{L}_{\mathsf{Free}}$ and the logbook index $\mathsf{L}_{\mathsf{Keys}}$ is updated with the new mapping of the user's verification key pk_U to the logbook address *addr*. Then, the home operator communicates with all other operators to ensure that everyone's logbook index is updated accordingly, and to obtain partial signatures on the mapping $(\mathsf{pk}_U, addr)$. Finally, the user receives the (combined) signature σ from the operators.

Protocol $\Pi_{\text{POBA}}(1^\lambda, n, t)$, Part 1

Building Blocks:
- Collision-resistant hash function H
- IND-CPA-secure threshold PKE scheme TPKE
- TS-UF-01-secure threshold signature scheme TSIG
- EUF-CMA-secure signature scheme SIG
- Simulation-extractable non-interactive zero-knowledge proof system ZK

Party State:
- Each operator O stores:
  - ZK CRS crs_{ZK}, TSIG parameters crs_{TSIG}
  - System encryption key ek, verification key vk
  - Corresponding secret key shares dk_i and sk_i
  - An operator keypair $(\text{sk}_O, \text{vk}_O)$
  - List $\mathsf{L}_{\text{Pending}}$ of pending entries
  - Value id_{last} for indexing pending entries
  - Shamir secret-share $\langle M_j \rangle$ of the oblivious memory M_j for each period j, all initially set to 0
- The operators store synchronized copies of:
  - List L_{Free} of free logbook addresses
  - Logbook index L_{Keys} that maps user public keys to logbook addresses
- Each user U stores:
  - ZK CRS crs_{ZK}, TSIG parameters crs_{TSIG}
  - System encryption key ek and verification key vk
  - Credentials $creds \coloneqq (\text{sk}_U, \text{pk}_U, addr, \sigma)$
  - List $\mathsf{L}_{\text{Receipt}}$ of receipts for logbook entries

Init:
- Input each O: $(\texttt{init})$
- Behavior (each O):
  1. Obtain and store crs_{ZK} and crs_{TSIG} from $\mathcal{F}_{\text{CRS}}$ (see Fig. 2.1a)
  2. Generate and store $(\text{sk}_O, \text{vk}_O) \leftarrow \text{SIG.Keygen}(1^\lambda)$
  3. Send $(\text{Keygen}, 1^\lambda)$ to $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$ `// Distributed key generation (see Fig. 2.3a)`
  4. Wait for output (ek, dk_i) from $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$
  5. Send $(\text{Keygen}, crs_{\text{TSIG}})$ to $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$ `// Distributed key generation (see Fig. 2.3b)`
  6. Wait for output (vk, sk_i) from $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$
  7. Store $\text{ek}, \text{vk}, \text{dk}_i, \text{sk}_i$
- Output O: $(\texttt{init}, \texttt{ok})$

Figure 4.5.: Protocol Π_{POBA} part 1, state and initialization.

Protocol $\Pi_{\text{POBA}}(1^\lambda, n, t)$, Part 2

UReg:
- Input U: $(\texttt{ureg}, ssid, pid_{HO})$
- Input HO: $(\texttt{ureg}, ssid, pid_U)$
- Input all other O': $(\texttt{ureg}, ssid)$
- Behavior:
  - HO: Send $(ssid\|1, pid_U, (\text{ek}, \text{vk}))$ to $\mathcal{F}_{\text{SMT}}$
  - U: Receive $(ssid\|1, pid_{HO}, (\text{ek}, \text{vk}))$ from $\mathcal{F}_{\text{SMT}}$
  - U: Store ek and vk
  - U: Obtain and store crs_{ZK} and crs_{TSIG} from $\mathcal{F}_{\text{CRS}}$
  - U: $(\text{sk}_U, \text{pk}_U) \leftarrow \text{SIG.Keygen}(1^\lambda)$
  - U: $\pi_{\text{sk}_U} \leftarrow \text{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, (\text{pk}_U), (\text{sk}_U))$
  - U: Send $(ssid\|2, pid_{HO}, (\text{pk}_U, \pi_{\text{sk}_U}))$ to $\mathcal{F}_{\text{SMT}}$
  - HO: Wait for output $(ssid\|2, pid_U, (\text{pk}_U, \pi_{\text{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$
  - HO: If $\text{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, (\text{pk}_U), \pi_{\text{sk}_U}) \neq 1$, abort
  - HO: Get next free logbook address $addr \leftarrow \mathsf{L}_{\text{Free}}.\text{pop}()$
  - HO: Add $(\text{pk}_U, addr, U)$ to L_{Keys}
  - $HO \rightarrow$ all O': $(ssid, \text{pk}_U, \pi_{\text{sk}_U}, addr, U)$
  - all O': $stmt \coloneqq (\text{pk}_U)$
  - all O': If $\text{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\text{sk}_U}) \neq 1$ or $addr \notin \mathsf{L}_{\text{Free}}$, abort
  - all O': Remove $addr$ from L_{Free}
  - all O': Add $(\text{pk}_U, addr, U)$ to L_{Keys}
  - all O': $\sigma_i \leftarrow \text{TSIG.PartSign}(crs_{\text{TSIG}}, \text{sk}_i, (\text{pk}_U, addr))$
  - all O' $\rightarrow HO$: $(ssid, \sigma_i)$
  - HO: $\sigma_i \leftarrow \text{TSIG.PartSign}(crs_{\text{TSIG}}, \text{sk}_i, (\text{pk}_U, addr))$
  - HO: Set T to be the first $t+1$ σ_i
  - HO: $\sigma \leftarrow \text{TSIG.Combine}(crs_{\text{TSIG}}, T)$
  - HO: Send $(ssid\|3, pid_U, (addr, \sigma))$ to $\mathcal{F}_{\text{SMT}}$
  - U: Receive $(ssid\|3, pid_{HO}, (addr, \sigma))$ from $\mathcal{F}_{\text{SMT}}$
  - U: Abort if $\text{TSIG.Vf}(\text{vk}, (\text{pk}_U, addr), \sigma) \neq 1$
  - U: Store $(\text{sk}_U, \text{pk}_U, addr, \sigma)$
- Output U: $(ssid, \texttt{ok})$
- Output HO: $(ssid, \texttt{ok})$

Figure 4.6.: Protocol Π_{POBA} part 2, user registration.

Signing both the user's public key and the address with the threshold signature scheme ensures that no one can impersonate honest users when creating entries: we require the user to prove knowledge of a signature of the entry under that public key to create a new entry, as well as knowledge of the threshold signature on $(\text{pk}_U, addr)$. Even if a minority of operators are corrupted, an adversary cannot create such a signature for $(\text{pk}'_U, addr)$ with a different public key but the address of an honest user, and the threshold signature

on $(\mathsf{pk}_U,\ addr)$ cannot be used to impersonate the honest user, since knowledge of sk_U is also required to create a new entry.

Protocol $\Pi_{\mathrm{POBA}}(1^\lambda, n, t)$, Part 3

CreateEntry:
- Input U: $(\texttt{createentry},\ ssid,\ period_U,\ entry_U,\ pid_O)$
- Input O: $(\texttt{createentry},\ ssid,\ period_O,\ entry_O)$
- Behavior:
  - U: Retrieve $(\mathsf{sk}_U,\ \mathsf{pk}_U,\ addr,\ \sigma)$, ek, vk and vk_{HO}, abort if those values are not yet stored
  - U: $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek},\ addr; r)$
  - U: $h \coloneqq H(period_U, ct_U, entry_U)$
  - U: $stmt \coloneqq (ct_U,\ \mathsf{ek},\ crs_{\mathsf{TSIG}},\ \mathsf{vk},\ h)$
  - U: $wit \coloneqq (\mathsf{pk}_U,\ \mathsf{sk}_U,\ addr,\ \sigma,\ r)$
  - U: $\pi \leftarrow \mathsf{Prove}(crs_{\mathsf{ZK}},\ \mathcal{R}_{\mathsf{CE}},\ stmt,\ wit)$
  - U: Send $(\texttt{send},\ ssid\|1,\ pid_O,\ (entry_U,\ ct_U,\ \pi))$ to $\mathcal{F}_{\mathrm{SMT}}^{\mathrm{os-auth}}$
  - O: Wait for output $(\texttt{send},\ ssid\|1,\ (entry_U,\ ct_U,\ \pi))$ from $\mathcal{F}_{\mathrm{SMT}}^{\mathrm{os-auth}}$
  - O: If $entry_U \neq entry_O$, abort
  - O: $h \coloneqq H(period_O, ct_U, entry_O)$
  - O: $stmt \coloneqq (ct_U,\ \mathsf{ek},\ crs_{\mathsf{TSIG}},\ \mathsf{vk},\ h)$
  - O: If $\mathsf{Vf}(crs_{\mathsf{ZK}},\ \mathcal{R}_{\mathsf{CE}},\ stmt,\ \pi) \neq 1$, abort
  - O: $id_{\mathsf{last}} \coloneqq id_{\mathsf{last}} + 1$
  - O: Add $(id_{\mathsf{last}},\ period_O,\ ct_U,\ entry_O,\ \pi)$ to $\mathsf{L}_{\mathsf{Pending}}$
  - O: $\sigma_{entry} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O,\ h)$
  - O: Send $(\texttt{respond},\ ssid\|1,\ (\sigma_{entry},\ \mathsf{vk}_O))$ to $\mathcal{F}_{\mathrm{SMT}}^{\mathrm{os-auth}}$
  - U: Wait for output $(\texttt{respond},\ ssid\|1,\ (\sigma_{entry},\ \mathsf{vk}_O))$ from $\mathcal{F}_{\mathrm{SMT}}^{\mathrm{os-auth}}$
  - U: Check $\mathsf{SIG.Vf}(\mathsf{vk}_O,\ h,\ \sigma_{entry})$
  - U: Store $receipt \coloneqq (entry_U,\ ct_U,\ r,\ \sigma_{entry},\ \mathsf{vk}_O)$ in $\mathsf{L}_{\mathsf{Receipt}}$
- Output U: $(ssid,\ \mathtt{ok})$
- Output O: $(ssid,\ \mathtt{ok},\ id_{\mathsf{last}})$

InsertEntry:
- Input O: $(\texttt{insertentry},\ ssid,\ period,\ id)$
- Input all other O': $(\texttt{insertentry},\ ssid)$
- Behavior:
  - O: Retrieve $(id,\ period,\ ct_U,\ entry,\ \pi)$ from $\mathsf{L}_{\mathsf{Pending}}$ and abort if no entry with $(id,\ period)$ exists
  - $O \to$ all O': $(ssid,\ period,\ ct_U,\ entry,\ \pi)$
  - all: $h \coloneqq H(period, ct_U, entry)$
  - all: $stmt \coloneqq (ct_U,\ \mathsf{ek},\ crs_{\mathsf{TSIG}},\ \mathsf{vk},\ h)$
  - all: If $\mathsf{Vf}(crs_{\mathsf{ZK}},\ \mathcal{R}_{\mathsf{CE}},\ stmt,\ \pi) \neq 1$, abort[a]
  - all: $ct_i \coloneqq \mathsf{TPKE.PartDec}(\mathsf{dk}_i,\ ct_U)$
  - all: Send $(\mathrm{Run},\ (ct_U,\ ct_i,\ entry), \mathsf{Share}M_{period})$ to $\mathcal{F}_{\mathrm{MPC(insert)}}$ and wait for output $(written)$
  - all: If $written$ = false output $(ssid,\ \mathtt{error})$
  - all: Otherwise output $(ssid,\ \mathtt{ok})$
- Output all O: $(ssid,\ \mathtt{ok/error})$

[a] This is not strictly required for semi-honest security

Figure 4.7.: Protocol Π_{POBA} part 3, entry creation and insertion.

CreateEntry As mentioned before, adding entries to a logbook is divided into two steps: First, the CreateEntry task is run between an (anonymous) user and an operator. In this step, a pending logbook entry is created and authenticated by the user. Specifically, the user and operator first agree on the content *entry* of the entry and the current period *period* (out-of-protocol). Then, the user (threshold) encrypts their logbook address *addr* as ct_U and proves the following in zero-knowledge:

- the user knows a valid (threshold) signature σ for their public key pk_U and logbook address *addr*
- the (threshold) encryption ct_U contains this logbook address *addr*
- the user knows the secret key sk_U corresponding to pk_U

Additionally, a hash of *period*, ct_U and the content of the entry is part of the statement, which thanks to the simulation-extractability ties the proof to the entry, preventing reuse of a proof generated for a different entry. As previously mentioned, this ensures that the user has previously registered an account and prevents an adversary from impersonating an honest user. The operator checks this proof, stores the pending entry in a list of pending entries, and provides the user with a receipt *receipt* for the entry, which contains a signature σ_{entry} on the above hash.

RAM-program insert$(\{ct^O, ct_i^O, msg^O\}, M)$

1. Set $ct \coloneqq ct^O$, $msg \coloneqq msg^O$ for the first operator O
2. Set $addr \coloneqq \mathsf{TPKE.Combine}(ct, (ct_i^{O_1}, \ldots, ct_i^{O_t}))$
3. Set $\mathsf{Log} \coloneqq M[addr]$
4. Set $written \coloneqq \mathsf{false}$
5. For $j = 0, \ldots,$ *maxentries*:
   - If $written = \mathsf{false} \wedge \mathsf{Log}[j] = \perp$:
     a) $\mathsf{Log}[j] \coloneqq msg$
     b) $written \coloneqq \mathsf{true}$
6. Write the updated logbook back: $M[addr] \coloneqq \mathsf{Log}$
7. Output *written*

Figure 4.8.: The RAM program insert to write log entries into the correct logbook.

InsertEntry The second step is performed later jointly by the operators, where the pending entries are actually written to the correct logbooks. This is done by each operator first locally performing a partial decryption of the user's encrypted logbook address and then running a secure multi-party protocol $\mathcal{F}_{\mathrm{MPC(insert)}}$ (see Figs. 4.3a and 4.8) that takes the pending entry along with the decryption shares as input, reconstructs the user's logbook address, and then writes the entry to the logbook with that address.

Doing partial decryption in the clear and only having to do the (additive) reconstruction of the plaintext from the partial decryptions in MPC gives a significant performance advantage over doing the whole decryption in MPC. Decoupling the (lightweight) interaction with the user from the execution of this secure multi-party protocol not only allows faster user interactions, but also to handle peaks higher than the throughput of the MPC, as long as the average remains below the throughput.

ProveEntry This task is used when a user wants to prove to an operator that they have made a certain logbook entry. Note that this operator intentionally learns the user's identity, the content of the entry, the period for which the entry was made, and the operator with which the entry was made. The basis for this task is the receipt *receipt* that the user received from CreateEntry as confirmation of their entry. The operator is given a period *period*, a message *msg*, a user public key pk_U, a logbook address *addr*, a (threshold) signature σ, an encrypted logbook address ct_U, an entry signature σ_{entry}, an operator verification key vk_O, and a zk proof π, and checks

- that there exists a logbook entry for the message *msg* in period *period* (for *some* user): for this, the operator checks the validity of the signature σ_{entry} (from the CreateEntry task) on the logbook entry *msg*, the period *period*, and the encrypted logbook address ct_U under the operator verification key vk_O
- whether the user who made the entry is a registered user: for this, the (threshold) signature σ (from the UReg task) on $(\mathsf{pk}_U,\ addr)$ is checked
- that the user who made the entry is the same user who is currently executing this task: to do this, the user proves to the operator in zero-knowledge that the encrypted logbook address ct_U contains *addr* and that they know the secret key associated with the public key pk_U of the user who made the entry

Protocol $\Pi_{\text{POBA}}(1^\lambda, n, t)$, Part 4

ProveEntry:
- Input U: $(\texttt{proveentry}, ssid, entry, period, pid_O)$
- Behavior:
  - U: Retrieve $(\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$, ek and vk
  - U: Retrieve $receipt \coloneqq (entry, ct_U, r, \sigma_{entry}, \mathsf{vk}_O)$ from $\mathsf{L}_{\text{Receipt}}$. If no such receipt exists, abort
  - U: $stmt \coloneqq (\mathsf{pk}_U, addr, ct_U, \mathsf{ek})$, $wit \coloneqq (r, \mathsf{sk}_U)$
  - U: $\pi \leftarrow \mathsf{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\text{PE}}, stmt, wit)$
  - U: Send $(ssid\|1, pid_O, (period, entry, \mathsf{pk}_U, addr, \sigma, ct_U, \sigma_{entry}, \mathsf{vk}_O, \pi))$ to $\mathcal{F}_{\text{SMT}}$
  - O: Wait for output $(ssid\|1, pid_U, (period, entry, \mathsf{pk}_U, addr, \sigma, ct_U, \sigma_{entry}, \mathsf{vk}_O, \pi))$ from $\mathcal{F}_{\text{SMT}}$
  - O: Check if vk_O is a valid operator key, else abort
  - O: $h \coloneqq H(period, ct_U, entry)$, $stmt \coloneqq (\mathsf{pk}_U, addr, ct_U, \mathsf{ek})$
  - O: If $\mathsf{SIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$, abort
  - O: If $\mathsf{SIG.Vf}(\mathsf{vk}_O, h, \sigma_{entry}) \neq 1$, abort
  - O: If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{PE}}, stmt, \pi) \neq 1$, abort
  - O: Find $(\mathsf{pk}_U, addr, U)$ in L_{Keys}
- Output O: $(ssid, \texttt{proveentry}, pid_U, entry, period)$

Analytics:
- Input each O: $(\texttt{analytics}, ssid, func, \{period_i\}, \{U_i\}, aux_O)$
- Behavior:
  - all: For each $U \in \{U_i\}$ lookup $addr_i$ in L_{Keys}
  - all: Send $(\text{Run}, ssid\|1, \{\langle M_j \rangle\}_{j \in \{period_i\}}, (\{addr_i\}, aux_O))$ to $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ and wait for output $(\{\langle M_j \rangle_i\}_{j<k}, output_i, round, sequence)$
  - all: Set $\texttt{result}_O \coloneqq output_i$
- Output each O: $(ssid, \texttt{result}_O)$

Figure 4.9.: Protocol Π_{POBA} part 4, proving existence of an entry and analytics.

If all checks succeed, the operator is convinced that this user indeed made the entry *msg* during period *period* with the operator whose public key is vk_O.

Analytics To perform analytics on the user logbooks, all operators have to agree on some admissible function *func* to evaluate. This function is then evaluated using the secure multi-party protocol (see Fig. 4.3b). If data from multiple periods should be used, then it is gathered from each ORAM independently and combined.

4.3.3. Security

First, we informally argue why our protocol satisfies the desired security properties: To create logbook entries, users need a threshold signature on their public key and their assigned logbook address. The security of TSIG implies that entries can only be created for registered users. Additionally, a proof of knowledge of the user's secret key is provided. This prevents anyone from creating logbook entries in the name of honest users, and since the zero-knowledge proof is non-malleable (implied by simulation-extractability) and the (hash of the) entry is part of the statement, it also prevents changing entries before they are inserted into the logbook. Furthermore, a logbook entry also requires the signature of the operator involved in order to be used in ProveEntry, and it will only end up in the logbook if the involved operator agrees and forwards it to the other operators. Thus, entries are only created if both user and operator agree. The logbook itself is secret-shared between

the operators, preventing manipulation of its contents. The use of threshold encryption for the logbook address and NIZK proofs during creation ensures that the user's identity remains hidden. When inserting entries, the secure computation functionality ensures that no information about the logbook involved is revealed. To later prove having created an entry, a user simply presents the signature they received if and only if they successfully completed a CreateEntry task for that message, along with the signature on their public key and logbook address. Computation of analytics functions is done by $\mathcal{F}_{\text{MPC}(func)}^{\text{ext}}$, which does not reveal anything except the desired output and the number of memory accesses, which we do not consider a privacy violation.[8]

Security is captured by the following theorem, which we formally prove in Section B.2.

Theorem 4.3.1 (informal). *Π_{POBA} UC-realizes $\mathcal{F}_{\text{POBA}}$ wrt. adversaries $\mathcal{A}$ that may statically corrupt any number of users maliciously and up to $t \leq \frac{n-1}{2}$ operators semi-honestly.*

4.4. Applications

POBA has various applications in areas where users should be able to perform transactions in an anonymous and unlinkable way, but operators still need to be able to analyze the collected transaction data. These applications include, but are not limited to, check-in/check-out based payment systems for public transportation, charging/discharging electric vehicle batteries in smart grids, coalition loyalty programs, and behavior-based car insurance.

4.4.1. Check-In/Check-Out Based Payment System for Public Transportation

As an exemplary application for POBA, we propose a check-in/check-out based payment system for public transportation. In the context of mobility-as-a-service (MaaS), it will become beneficial to offer user-friendly systems that incorporate the services of several mobility providers. Therefore, we sketch here a payment system for multimodal public transportation where a customer's trip data is collected and analyzed in a privacy-preserving way. Note that users are anonymous during trips, but spot checks are identifying.

Registration When registering, users choose a home mobility provider to create an account with.

[8] While in theory a function could be designed in a way that the number of memory accesses encodes some private information, such behavior can be interpreted as part of the output and honest operators should not agree to compute them.

Check-in/Check-out A check-in/check-out-based system allows users to board transportation vehicles without first purchasing a ticket. Instead, the fare is calculated at a later time. More precisely, when using trains, for example, users submit a 'Check-in' message to their logbook upon entering the train, and a 'Check-out' message upon exiting. Fare calculation is deferred until the end of the billing period, e.g., the end of the month. This enables the use of flexible tariffs, such as combining multiple individual trips into a cheaper daily flat rate. The check-in and check-out process is handled by the CreateEntry task with the mobility provider with whom the trip takes place. Communication may actually be with a subsidiary of the mobility provider (e.g., a device in a train). A logbook entry may look like this: $msg \coloneqq (\texttt{check\_in/check\_out}, train\_type, train\_number, terminal\_station, current\_station, current\_time)$. In practice, all communication required for check-in/check-out can be achieved locally through short-distance wireless communication such as NFC or Bluetooth LE and requires neither the provider device nor the user device to have internet connectivity.

Spot Checks In public transportation systems without physical barriers to control access, ticket inspectors are commonly used to realize spot checks for passengers. This can be done in our system with the ProveEntry and CreateEntry tasks.

The customer first utilizes the ProveEntry task to prove that a message of type $msg \coloneqq (\texttt{check\_in}, train\_type, train\_number, terminal\_station, entry\_station, entry\_time)$ is in the database that matches the current train and a timestamp $entry\_time$ that is reasonably close to the current time. The ticket inspector[9] is now convinced that the user has a valid account at one of the operators and has previously checked-in to this train. Note that the ticket inspector learns the contents of the message msg, but everything except the entry station is already known by context.

To ensure that the user has not already checked out, CreateEntry is used: The ticket inspector and the user create a logbook entry for the message $msg \coloneqq (\texttt{inspection}, train\_type, train\_number, terminal\_station, current\_station, current\_time)$. The actual check that a user has not checked out yet is then postponed to the invoice calculation. In the case that the user had already checked out before the spot check, a fine can be added to the user's invoice during invoice generation.

Invoice Generation At the end of a billing period, for each user an invoice is generated, utilizing the Analytics task. There, all logbook entries for a user for the current billing period are loaded. Then they are checked for validity: Are there always matching `check_in` and `check_out` entries? Are `inspection` entries in between appropriate `check_in` and `check_out` entries? If all is valid, the invoice can be generated. If not, a fine is added to the invoice. Then, the invoice (consisting of the aggregate sum for the month and any extra fines with information about the respective spot-check) is output

[9] We assume that the ticket inspector acts as a subsidiary of the train's operator

to the respective home provider. Note that each user only gets an invoice from their home provider, regardless of the mobility providers they took trips with.

Clearing between Providers After issuing invoices to all users, operators initiate their clearing phase. The trip fees must be redistributed to the mobility providers with whom the trips were actually taken, as users pay their invoices to their home operators. To achieve this, the system also calculates during each invoice generation which trips were made with which mobility provider. The revenue for each mobility provider is then written to an array, secret-shared, and each mobility provider receives a share as output. When the next user's invoice is generated, all operators input their current share as auxiliary input *aux*, the MPC reconstructs the revenue array, updates it, and then secret-shares and outputs it again. Once all invoices for the current period have been created, the operators can initiate the actual clearing. For this, the shares are input into the MPC again, which then calculates the distribution of revenues among providers.

Route Load Analysis The Analytics task can also be used by mobility providers to analyze their route load to answer questions like "On which routes do more trains need to be deployed, where can trains be reduced, etc.?" By accessing data from all mobility providers, these analytics can take into account information beyond the boundaries of the provider's own network. This means that connections between different transportation networks can be better planned, resulting in a better overall experience for users.

4.4.2. Charging and Discharging Electric Vehicle Batteries in V2G Scenario

In the future smart grid, vehicle-to-grid (V2G) systems can use the batteries of plugged-in electric vehicles (EVs) as distributed energy storage. In this scenario, users are EV owners who want to charge their EVs at charging stations owned by different electricity providers. Operators are different electricity providers that own the EV charging stations. POBA can be used here as a privacy-preserving payment system for charging/discharging operations that handles the billing process between users and electricity providers and between the electricity providers themselves.

Registration When registering for the system, users select an electricity provider to create an account with. That electricity provider becomes the user's home operator.

Charging/Discharging EVs When an EV is charged (or discharged), a logbook entry is created with the electricity provider the charging station belongs to. Several pieces of information about this (dis-)charging process are written to the logbook, so the entry might look like this: $msg \coloneqq (\texttt{charge/discharge}, station\_id, date, start\_time, end\_time, kWh, price)$.

Invoice Generation & Clearing between Providers As in the check-in/check-out-based payment system for multimodal public transportation described earlier, the Analytics task can be used to generate invoices for individual users and to handle the clearing between the electricity providers.

Possibility of Complaint If a user discovers any discrepancies in their invoice, they can use the ProveEntry task to file a complaint with their home operator and clarify the issue. For example, if a discharge transaction, which would benefit the user monetarily, is missing from the invoice, the user can prove to their home operator that this transaction actually took place and should be included in the invoice. This discrepancy may occur if a logbook entry was created, but for some reason never inserted into the actual logbook.

Power Consumption Analysis The Analytics task can also be used to analyze past power consumption across the grid and to forecast future power consumption in specific regions or time periods. Such forecasts are important for ensuring the continued stability of the power grid.

4.5. Evaluation and Benchmarks

To evaluate the practical feasibility of POBA, we implemented our protocol and measured the execution times of the different interactions. We ran experiments for 3, 5 and 9 operators, each running on a server with an Intel® Xeon® E-2388G CPU @ 3.20 GHz and 64 GB RAM and connected via a 10 Gbit/s switch. We implemented the full protocol, but since user networking is difficult to control, we ignore the communication time between users and operators in our results.

4.5.1. Implementation Details

Our implementation is written in C++ and available at `https://github.com/kastel-security/poba`. To achieve efficient zero-knowledge proofs, we use the Groth–Sahai [GS08; EG14] proof system, which enables efficient proofs for pairing-product equations. To ensure all relations used in our protocol can be expressed as such pairing-product equations, we instantiate our building blocks in a compatible way over the pairing-friendly Barreto-Naehrig curve BN-382 [BN06], which provides roughly 128 bit of security [APR21], using the RELIC toolkit v.0.7.0 [AG20] as our arithmetic backend. To make the Groth–Sahai proof system simulation-extractable, we use the standard transformation as described in e.g. [HJ12], with the one-time signature described there along with BLS signatures for the simulation trapdoor. The other building blocks are instantiated as follows: For TPKE, we chose standard IND-CPA secure threshold ElGamal encryption over $\mathbb{G}_1$. For SIG, we use BLS signatures [BLS01] over $\mathbb{G}_1$. For TSIG, we use the structure-preserving threshold signature scheme of [Mit+24]. When signing and encrypting the logbook addresses of

users, we use the Koblitz encoding scheme [Kob87] to embed them in the x-coordinate of a point in $\mathbb{G}_1$. To implement $\mathcal{F}_{\text{MPC}(\cdot)}$ and $\mathcal{F}^{\text{ext}}_{\text{MPC}(\cdot)}$, we use the MP-SPDZ framework [Kel20] in the semi-honest honest majority setting with Shamir sharing. For better efficiency during entry insertion, we split the computation into two parts, which are run with different parameters: First, the partial decryptions of ct_U are combined in an instance MPC_1 over the field $\mathbb{F}_p$ over which BN-382 is defined. This allows us to obtain Shamir shares of the x-coordinate which encodes the logbook address. We then use the share conversion method described in [DT08] to convert them to Shamir shares over a field $\mathbb{F}_q$ with a 128 bit prime q chosen by MP-SPDZ. The rest of the computation is then done in an instance MPC_2 over $\mathbb{F}_q$, which uses PathORAM [KS14] for oblivious memory accesses. This splitting is done so that most of the computation, including the expensive ORAM accesses, does not have to be done over a 382 bit prime field but instead can be done in a smaller and MPC-friendly field.

Additionally, we also implemented the InsertEntry protocol for the special case of $n = 3, t = 1$ using GigaDORAM [Fal+23] for memory accesses. Concretely, we still make use of the MP-SPDZ instance MPC_1 to combine the partial decryptions, but then use the emp-toolkit [WMK16] as MPC_2, which is the MPC-backend of GigaDORAM. Since GigaDORAM uses replicated XOR secret sharing, we now need to perform bit-decomposition in MP-SPDZ to convert shares between these MPC-instances. Since this now represents the majority of the time spent in MPC_1 and involves many rounds of communication, we perform this step on batches of 20 entries together. To represent our logbook in GigaDORAM, which only supports memory cells with 64 bit, we assign $100\times\frac{\text{entry-size}}{64}$ consecutive addresses to each logbook, encrypting the starting address of this block in the ciphertext from the user. For our concrete example, logbook entries consist of 6 32 bit values, so we assigned 300 addresses to each logbook. To avoid having to perform 300 oblivious reads and 300 writes per insert, we store a pointer to the first free entry at the start of the logbook. Insertion then works by reading that pointer, calculating the addresses of the 3 relevant memory cells from it (i.e., using the pointer, pointer+1 and pointer+2) and updating the stored pointer to the value of the next free entry (i.e., old pointer +3). Then the entry gets written to the three relevant memory cells, resulting in a total of 1 oblivious read and 4 oblivious writes.

4.5.2. Results

We use the check-in/check-out based payment system for public transportation from Section 4.4.1 as an application to benchmark. Thus, our logbooks contain 100 slots of 6×32 bit each to store check-in/check-out messages. We benchmark ORAMs of size 2^{15} ($\approx$32 000) and 2^{20} ($\approx$1 million) entries, which results in the same number of maximum registered users in the protocol. As analysis function, we use the invoice generation.

We averaged our results over 100 interactions for UReg and CreateEntry. For InsertEntry, we averaged over 10 runs with a batchsize of 100 entries, where everything besides

Table 4.1.: MP-SPDZ with PathORAM: Average time (in ms) and transmitted data (in kB/MB) to complete the InsertEntry. Pretasks refers to the time spent verifying the zero-knowledge proofs and performing partial decryption, Conv. refers to the conversion of shares from the 382 bit prime used in MPC_1 to the 128 bit prime of MPC_2, and MPC_1 and MPC_2 to the time spent within the respective MPC protocol. The setting differentiates between the number of operators n and the number of users #. All times are amortized over 100 entries.

Setting	Pretasks	MPC_1		Conv.	MPC_2	
n=3, #=2^{15}	41.9 ms	0.1 ms	1.1 kB	0.0 ms	550 ms	12.2 MB
n=3, #=2^{20}	42.4 ms	0.1 ms	1.1 kB	0.0 ms	713 ms	14.3 MB
n=5, #=2^{15}	42.0 ms	0.5 ms	3.7 kB	0.1 ms	1074 ms	31.0 MB
n=5, #=2^{20}	42.5 ms	0.5 ms	3.7 kB	0.1 ms	1347 ms	35.6 MB
n=7, #=2^{15}	42.0 ms	0.9 ms	7.7 kB	0.1 ms	1804 ms	56.2 MB
n=7, #=2^{20}	42.1 ms	0.9 ms	7.7 kB	0.1 ms	2212 ms	63.8 MB
n=9, #=2^{15}	42.1 ms	1.4 ms	13.3 kB	0.2 ms	2791 ms	88.1 MB
n=9, #=2^{20}	42.3 ms	1.4 ms	13.3 kB	0.2 ms	3377 ms	99.1 MB

accessing ORAM is parallelized. From this, we calculated the amortized cost of inserting a single entry.

First, as expected, the running times of protocols involving users, i.e., UReg, CreateEntry and ProveEntry, are very fast. CreateEntry and ProveEntry are independent of the number of operators or other users involved. UReg gets slightly slower with more operators, with 238 ms for 3 operators down to 278 ms for 15 operators, ignoring communication time between user and home operator (but including communication between operators). This allows for account creation to finish in under one second when performed over the internet. Creating a new entry takes 105 ms (again ignoring communication time between user and home operator), which again allows the interaction including communication over NFC or BLE to finish in one to two seconds, a time common for other NFC-based transactions. Proving an entry was created (e.g., during spot-checks) takes 0.22 s and consists of a single message from user to inspector which could be transmitted through scanning of a QR-code or again NFC. Thus, we achieved our goal of having very efficient client-facing protocols.

On the operator end, both tasks can also be parallelized to improve throughput: user registration only requires synchronizing free logbook addresses or assigning pools of free addresses to each instance, and CreateEntry and ProveEntry require no synchronization.

For InsertEntry and Analytics, the runtime depends on both the number of operators and the number of registered users (i.e., the size of the ORAM). Results for $n = 3, 5, 9$ operators and number of registered users $\# = 2^{15}, 2^{20}$ when using MP-SPDZ are shown in Table 4.1. For all results, we used the maximum corruption threshold, namely $t = \frac{n-1}{2}$. The majority of the time spent in the second MPC protocol is reading and writing the logbook from memory: For $n = 9$ operators and 2^{15} users, out of the 2.79 s for MPC_2, 1.36 s are spent

Table 4.2.: MP-SPDZ with PathORAM and more than 10 parties: Average time (in ms) and transmitted data (in kB/MB) to complete the InsertEntry. Pretasks refers to the time spent verifying the zero-knowledge proofs and performing partial decryption, Conv. refers to the conversion of shares from the 382 bit prime used in MPC_1 to the 128 bit prime of MPC_2, and MPC_1 and MPC_2 to the time spent within the respective MPC protocol. The setting differentiates between the number of operators n and the number of users #. All times are amortized over 100 entries. Note that compared to Table 4.1, this uses servers with roughly half the performance.

Setting	Pretasks	MPC_1		Conv.	MPC_2	
n=11, #=2^{15}	42.0 ms	2.0 ms	20.2 kB	0.4 ms	8468 ms	126.2 MB
n=11, #=2^{20}	42.1 ms	2.0 ms	20.2 kB	0.4 ms	10 340 ms	141.2 MB
n=15, #=2^{15}	42.2 ms	3.1 ms	38.6 kB	2.3 ms	16 407 ms	221.9 MB
n=15, #=2^{20}	42.2 ms	3.1 ms	38.6 kB	2.3 ms	19 617 ms	246.1 MB
n=19, #=2^{15}	42.2 ms	5.3 ms	62.9 kB	28.4 ms	23 280 ms	344.5 MB
n=19, #=2^{20}	42.6 ms	4.9 ms	62.9 kB	28.3 ms	28 256 ms	380.3 MB

reading the logbook from memory, and 1.35 s are spent writing the updated logbook back. Updating the logbook with the new entry, which involves iterating over all 100 slots to find the first empty one, only takes the remaining 0.08 s.

For InsertEntry, it is possible to parallelize the verification of the zero-knowledge proof and the partial decryption, and the reconstruction of the shared logbook address from the partial decryption, i.e., execution of MPC_1. MPC_2 must be executed sequentially because PathORAM does not support parallel memory access. Thus, the bottleneck for throughput is the time required for MPC_2.

We performed some additional benchmarks with $n = 11$, 15 and 19 servers. This uses servers equipped with Intel® Xeon® E-2288G CPU @ 3.70 GHz and 32 GB RAM for operators 11-19 instead of the ones with Intel® Xeon® E-2388G CPU @ 3.20 GHz and 64 GB RAM used for operators 1-10. Importantly, these have significantly worse performance across all evaluated tasks, taking roughly twice the time for UReg, CreateEntry and all the InsertEntry tasks if run with 3-9 parties solely on these servers. While it would be interesting to find the reason for this performance difference, we did not investigate this. Table 4.2 shows the results for these extended benchmarks. Note that the time for share conversion increases drastically because our naive approach has quadratic communication complexity, so employing techniques similar to those used in MP-SPDZ to reduce this to linear communication complexity would improve this. However, given this is clearly not the bottleneck, we did not further optimize this.

Overall, ORAM in MP-SPDZ currently does not scale well with an increasing number of parties, with the throughput for 9 operators already severely limiting the number of users. Further increasing the number of operators to e.g., $n = 19$ reduces throughput to roughly 3000 entries per day.

Table 4.3.: InsertEntry with GigaDORAM and $n = 3$ operators. MPC_1 is the (amortized) time per entry spent in MP-SPDZ to combine the partial decryptions to obtain the logbook address and convert the resulting Shamir-sharing into xor-sharing required for GigaDORAM. MPC_2 is the time per entry spent in emp-toolkit to write the entry into ORAM. $\sum$ is the total (amortized) time for the whole InsertEntry protocol, additionally consisting of verification of the zero-knowledge proof and partial decryption of the ciphertext.

Logbooks	MPC_1	MPC_2	$\sum$
2^{15}	80 ms	39 ms	126 ms
2^{20}	80 ms	42 ms	128 ms

For the special case of $n = 3$, optimized protocols for distributed ORAM significantly increase throughput. When using GigaDORAM [Fal+23] and the emp-toolkit (on which the implementation of GigaDORAM is based) for MPC_2, the time to access ORAM decreases drastically, with the whole of MPC_2 only taking 39 ms per entry with 2^{15} logbooks and 42 ms with 2^{20} logbooks, see Table 4.3. While performance seems to allow for much greater numbers of logbooks and logbook sizes, the current implementation of GigaDORAM limits the size of the memory to $< 2^{30}$, and with each of our logbooks taking $300 \approx 2^9$ our implementation is limited to $\approx 2^{20}$ logbooks. The running time of MPC_1 on the other hand increases to 1.6 s for 20 entries, or on average 80 ms per entry. Performing the required steps sequentially, our implementation takes ≈0.13 s per entry, allowing ≈650 000 entries per day. However, only MPC_2 needs to be sequentialized (since GigaDORAM also does not support parallel accesses), whereas verification of the zero-knowledge proof, partial decryption and MPC_1 can be parallelized across multiple servers per operator. This way, the 0.042 s for MPC_2 limit the throughput, allowing over two million entries per day with a million registered users.

A public transport operator serving a rural area with a population of around half a million told us that they have around 370 000 registered users, peaking at around 10 million trips in a rainy November, with an average of around 700 000 entries (two for each trip) per day. Thus, a consortium of three operators could already use our system in this setting. On the extreme end however, Transport for London reports 26 million daily trips for the greater London area, which is currently far beyond the limit of our protocol.

4.6. Achieving Malicious Security

In this section, we explain how to extend our protocol Π_{POBA} to achieve security against malicious operators.

The main obstacle towards this goal is the practical efficiency of maliciously-secure MPC, especially when incorporating secure memory accesses. On a theoretical level however, it is easy to extend our functionalities $\mathcal{F}_{\text{MPC}(f)}$ (cp. Fig. 4.3a) and $\mathcal{F}^{\text{ext}}_{\text{MPC}(f)}$ (cp. Fig. 4.3b) to provide security against malicious parties: All that is required is to ensure that the shares of

the memory that are input are a consistent sharing of the memory. Additionally, our RAM-program insert (cp. Fig. 4.8) needs to ensure that the partial decryptions provided as input are valid. To this end, we require our secret-sharing scheme S, as well as the secret-sharing used for TPKE to have an additional property which we call *weak robustness*.

A secret-sharing scheme is called *robust* if reconstruction always outputs the correct secret as long as at least $t + 1$ of the shares provided to the reconstruction algorithm are honest shares output by the sharing algorithm, regardless of how the other $n - t - 1$ are manipulated. Since MPC protocols in our setting can only achieve security-with-abort anyway, we only require a weaker notion, which states that reconstruction either outputs the correct secret if all n shares provided are honest shares output by the sharing algorithm, or a special output $\perp$ otherwise. Formally, this is captured in Definition 4.6.1.

Definition 4.6.1 (Weakly robust secret-sharing)**.** We call a secret-sharing scheme S *weakly robust* if it holds that the output of $\mathsf{S.Recon}(\langle m\rangle_1, \ldots, \langle m\rangle_n)$ is m if and only if there exists randomness such that $(\langle m\rangle_1, \ldots, \langle m\rangle_n)$ is the output of $\mathsf{S.Share}(m)$, and otherwise S.Recon outputs $\perp$.

For example, Shamir's secret-sharing scheme fulfills this requirement: S.Recon can check whether all n shares provided lie on the same degree-t polynomial g and output $g(0)$ in that case and $\perp$ otherwise. To fulfill full robustness however it would require that $n > 3t + 1$. Similarly, replicated secret-sharing fulfills this requirement. Since our implementation of TPKE is based on Shamir's secret-sharing, it also fulfills this requirement.

Now, we can simply extend $\mathcal{F}_{\mathrm{MPC}(f)}$ and $\mathcal{F}^{\mathrm{ext}}_{\mathrm{MPC}(f)}$ to verify that the output of reconstruction is not $\perp$, as shown in Figs. 4.10a and 4.10b, and similarly extend insert to ensure that all operators input the same ciphertext and log entry as well as consistent decryption shares, as shown in Fig. 4.11.

In our implementation, we then also need to ensure that passing information between MPC_1 and MPC_2 occurs in a way that is secure against malicious adversaries. This is achieved by generating a fresh Shamir-sharing of the address in MPC_1 and outputting a share to every operator. This is important, as directly using the shares from MP-SPDZ would circumvent the MAC-check that ensures correctness of the computation. Then we again use the share-conversion method of Damgard and Thorbek [DT08], which turns a valid Shamir-sharing of some secret m with $m \leq q$ in $\mathbb{F}_p$ into a valid Shamir-sharing of the same secret in $\mathbb{F}_{p'}$ given that $q < p' < p$. Then we can verify that all n shares lie on the same degree-t polynomial before reconstructing in MPC_2 again. Again, since MP-SPDZ only guarantees security-with-abort, it is fine that our protocol aborts if inconsistent shares are input, as the malicious party inputting an inconsistent share to cause the protocol to abort could achieve this abort anyway.

Many other parts that are important for malicious security, while not strictly necessary for semi-honest security, are already present in our protocol since their overhead is negligible compared to the cost of MPC. These are:

Functionality $\mathcal{F}_{\mathbf{MPC}}(f)$

Parameters: A next-step function NS, number of participating parties n

- Upon receiving (Run, $\langle M \rangle$, $input_i$) from all parties P_i, $i < n$:
  1. $M = \mathsf{S.Recon}(\langle M \rangle)$
  2. If $M = \perp$, abort
  3. Initialize $\mathsf{st}_{\mathsf{NS}} \coloneqq (\mathsf{start}, (input_1, \ldots, input_n))$, $data \coloneqq 0$ and $round \coloneqq 0$
  4. While $\mathsf{st}_{\mathsf{NS}} \neq (\mathsf{stop}, (output_1, \ldots, output_n))$, do:
     a) Run $(\mathsf{st}_{\mathsf{NS}}, I) \leftarrow \mathsf{NS}(\mathsf{st}_{\mathsf{NS}}, data)$
     b) If $I = (\mathsf{read}, addr, \perp)$, set $data \coloneqq M[addr].val$
     c) Else if $I = (\mathsf{write}, addr, val)$, set $data \coloneqq M[addr].val$ and update $M[addr].val = I.val$
     d) Update $round \coloneqq round + 1$
  5. $\langle M^{\mathrm{new}} \rangle \leftarrow \mathsf{S.Share}(M)$
  6. Output $(\langle M \rangle_i, output_i, round)$ to each party P_i, $i < n$.

(a) Functionality $\mathcal{F}_{\mathrm{MPC}(f)}$ for securely evaluating RAM programs.

Functionality $\mathcal{F}^{\mathbf{ext}}_{\mathbf{MPC}}(f)$

Parameters: A next-step function NS, number of participating parties n

- Upon receiving (Run, $\{\langle M_j \rangle\}_{j<k}$, $input_i$) from all parties P_i, $i < n$:
  1. For each $j < k$: $M_j = \mathsf{S.Recon}(\langle M_j \rangle)$
  2. If $M_j = \perp$ for any $j < k$, abort
  3. Initialize $\mathsf{st}_{\mathsf{NS}} \coloneqq (\mathsf{start}, (input_1, \ldots, input_n))$, $data \coloneqq 0$, $round \coloneqq 0$ and $sequence = ()$
  4. While $\mathsf{st}_{\mathsf{NS}} \neq (\mathsf{stop}, (output_1, \ldots, output_n))$, do:
     a) Run $(\mathsf{st}_{\mathsf{NS}}, I) \leftarrow \mathsf{NS}(\mathsf{st}_{\mathsf{NS}}, data)$
     b) If $I = (\mathsf{read}, index, addr, \perp)$:
        i. Ensure $index < k$, otherwise abort
        ii. Set $data \coloneqq M_{index}[addr].val$
     c) Else if $I = (\mathsf{write}, index, addr, val)$
        i. Ensure $index < k$, otherwise abort
        ii. Set $data \coloneqq M_{index}[addr].val$
        iii. Update $M_{index}[addr].val = I.val$
     d) Update $round \coloneqq round + 1$ and $sequence \coloneqq (sequence, index)$
  5. For each $j < k$: $\langle M_j^{\mathrm{new}} \rangle \leftarrow \mathsf{S.Share}(M_j)$
  6. Output $(\{\langle M_j \rangle_i\}_{j<k}, output_i, sequence)$ to each party P_i, $i < n$.

(b) Functionality $\mathcal{F}^{\mathrm{ext}}_{\mathrm{MPC}(f)}$ for securely evaluating extended RAM programs.

Figure 4.10.: Functionalities for secure computation with security against *malicious* parties. Changes compared to the semi-honest version are highlighted.

RAM-program $\mathsf{insert}(\{ct^O, ct_i^O, msg^O\}, M)$

1. If for all $O \in \mathcal{O}$ ct^O and msg^O are the same, set those as ct and msg, otherwise output $\perp$
2. Set $addr \coloneqq \mathsf{TPKE.Combine}(ct, (ct_i^{O_1}, \ldots, ct_i^{O_n}))$ // Needs all n shares now
3. If $addr = \perp$, output $\perp$
4. Set $\mathsf{Log} \coloneqq M[addr]$
5. Set $written \coloneqq \mathsf{false}$
6. For $j = 0, \ldots, maxentries$:
   - If $written = \mathsf{false} \wedge \mathsf{Log}[j] = \perp$:
     a) $\mathsf{Log}[j] \coloneqq msg$
     b) $written \coloneqq \mathsf{true}$
7. Write the updated logbook back: $M[addr] \coloneqq \mathsf{Log}$
8. Output $written$

Figure 4.11.: The RAM program insert to write log entries into the correct logbook with security against *malicious* parties. Changes compared to the semi-honest version are highlighted.

- Verification of the signature σ_{entry} received from the operator during CreateEntry on the user side
- Verification of the NIZK-proof during InsertEntry on the side of all operators

However, even in the semi-honest setting, these checks provide additional resilience against accidental faults. The main changes required to our protocol for malicious security are 1. ensure malicious operators cannot lie about the public keys of the system—this can be solved through a PKI and an aggregate signature of all operators on the verification and encryption key of the scheme 2. ensure user registration is handled consistently

across all operators—this requires consensus 3. ensure each logbook entry created through CreateEntry can only be inserted once—this can easily be realized by having each operator keep a list of all entries that have already been inserted per period

For user registration, it is important to ensure that each logbook address can only be assigned to a single user. For this, all operators need to be in agreement on which addresses are associated with a public key, and which addresses are still free. We can achieve this by having the home operator send the message containing the users public key, zero-knowledge proof and the chosen logbook address over an atomic broadcast channel instead of point-to-point channels, and also having all operators send their partial signatures back over atomic broadcast instead of a point-to-point channel. Additionally, operators should wait for all atomic broadcasts to finish before participating in another invocation of UReg.

While this has a significant impact on the performance of UReg, users only need to register once. Compared to the high overhead of maliciously secure MPC, this has little impact on the overall protocol efficiency.

Benchmarks for Malicious Security To evaluate the impact of malicious security on performance, we configured MP-SPDZ to use malicious Shamir instead of semi-honest Shamir as backend for both MPC_1 and MPC_2 and added the additional checks detailed above to the circuits. For tasks not involving MPC, there is no change, i.e., the user-facing performance stays the same. However, the performance of InsertEntry and thus the throughput of the system severely degrades. We show in Table 4.4 our results by using the same parameters and hardware setup as in Table 4.1. However, MP-SPDZ needs more than 10 times the amount of data to be transmitted and the computation time also increases by more than a factor of 10 to achieve malicious security. As Table 4.4 shows, already for the smallest setting of $n = 3$ and $\# = 2^{15}$ our malicious security protocol takes 10s to insert an entry. This translates to a throughput of roughly 8500 entries per day, which is only useful in very limited scenarios. Going to 5 operators degrades performance to almost a minute per entry, clearly showing the current limits of malicious security.

As can be seen by the stark contrast between the performance of MPC_1 and MPC_1, the main bottleneck lies in the maliciously secure handling of oblivious RAM inside MP-SPDZ. Further advances in maliciously-secure oblivious RAM for MPC are required to make malicious instantiations of our protocol feasible.

As GigaDORAM does not support malicious security, we did not benchmark the special case of $n = 3$ in that setting.

Table 4.4.: MP-SPDZ with PathORAM for malicious security: Average time (in ms/s) and transmitted data (in kB/MB) to complete the InsertEntry task, where $\sum$ refers to the total time of InsertEntry, and MPC_1 and MPC_2 to the (amortized) time spent within the respective MPC protocol for InsertEntry. The setting differentiates between the number of operators n and the number of users #.

Setting	InsertEntry				
	MPC_1		MPC_2		$\sum$
n=3, #=2^{15}	1 ms	36 kB	10 s	340 MB	10 s
n=3, #=2^{20}	1 ms	36 kB	12 s	410 MB	12 s
n=5, #=2^{15}	6 ms	100 kB	53 s	840 MB	53 s
n=5, #=2^{20}	6 ms	100 kB	55 s	1000 MB	55 s

5. Comparison and Outlook

In this chapter, we take a comparative look at the two solutions presented so far. Then, we point to some interesting open problems and ideas to further improve the practicability.

5.1. Comparison

There are many factors to consider when choosing between data storage on user devices versus operator devices.

One such factor is the question of data sovereignty: As long as the collected data is only stored on user devices, they are in control of the usage of that data. For each individual computation involving their data, they can choose whether they want to participate. (Of course, this comes with the caveat that those guarantees only hold as long as the parties involved in the actual computation, i.e., the proxy and operator in the case of PUBA, do not collude.) When the data is stored with the system operators, users can only decide whether they want to participate in the system as a whole, and thus trust the system to adequately restrict analysis of their data. Subsequently, the determination of which computations offer an adequate level of privacy is exclusively within the purview of the system operators as a collective. While it may be reasonable to hypothesise that not all operators will collude to violate laws and regulations such as the GDPR, it is more straightforward to envisage analytical results that fall within the provisions of such regulation but to which a user might not consent if presented with the choice.

Another consideration is that of data protection: In the context of data storage on user devices, each user assumes responsibility for safeguarding their data against both theft and loss. In many use cases for PUBA, a system may require penalties for users who do not participate in required analytical computations as a deterrent against malicious behavior. Losing the device or the data on it can have serious consequences, for example in the context of post-payment systems. Conversely, loss of data can negatively impact users even without such punishments, as in a prepaid payment system, as presented in Section 3.5.1, where this would result in loss of the current balance.

When data is stored on operator devices, the responsibility for protecting it shifts to companies with more experience and resources. Given the distributed and secret-shared nature of data storage in POBA, as long as a sufficient number of operators can keep their portion of the data confidential and accessible, the system as a whole guarantees the confidentiality and accessibility of all user logbooks. Even if one (or even up to the

corruption threshold t many) operators completely lose all their data, the system can recover and continue operation without loss of data.

A third consideration is the convenience of use: For systems based on regular analyses of user data, which are either mandatory for participation in the system or designed to be privacy-preserving, such as the calculation of aggregate statistics with suitable levels of k-anonymity or differential privacy, requiring active user interaction is inconvenient.

Finally, functionality is obviously a major concern: In POBA, interactions with users cannot make decisions based on the content of the user logbook. For example, this precludes the fraud-detection system described in Section 3.5.1. Conversely, logbook sizes in PUBA are limited by the available hardware resources and will typically be limited by the capabilities of the least powerful supported hardware.

5.2. Future Work/Outlook

The evaluation of the two systems described in this thesis presented some interesting open problems.

PUBA PUBA is currently limited to a single system operator, and requires a non-colluding proxy for analysis computations. Thus, an important open problem is to extend it to a setting with multiple system operators. This will most likely include threshold (for honest majority) or aggregate/multi-signatures (for no honest majority) signatures on logbooks instead of the normal signature, thus requiring all protocols to involve all system operators. Additionally, the verification of correct logbook updates will have to be extended in a way that lets all operators verify the update was applied correctly, i.e., even when the user and the operator responsible for the new entry are both corrupted, changes should be limited to those that could result from legitimate interactions. For example, in a prepaid payment system, a malicious vendor colluding with a user should not be able to arbitrarily increase the user's balance.

POBA As seen in Section 4.5, the use of oblivious RAM presents the main bottleneck. Recall that the reason for oblivious RAM was to remove the link between cleartext logbook entry during CreateEntry and accessing secret-shared logbooks during Analysis. Another way of removing this link are mixnets, as for example deployed in electronic voting systems [Adi08]. If all pending entries for a period are randomized through a mixnet before decrypting the user id, this achieves almost the same unlinkability guarantees as the oblivious RAM as long as the memory patterns of logbook reads during analysis do not depend on sensitive data, for example by simply reading each relevant logbook once. The only additional leakage then would be the number of entries in each logbook. If this is a problem, an extra step of oblivious sorting with dummy-entries to fill up each logbook can be used to also hide this information, at the cost of extra computation.

We conjecture that such a system could achieve a much larger throughput and scale better in both the number of system operators as well as maximum logbook sizes per period. Particularly, if leakage on the size of each users logbook is acceptable, the system only scales in the actual number of entries per period, not in the maximum allowed number of entries per user, and logbook sizes can be unlimited.

Combining Both Approaches In addition to the possible improvements to both presented systems individually, several of the trade-offs described in Section 5.1 can be mitigated by a system that combines storage on user devices with storage on operator devices. One option is to keep especially sensitive data on the user device to provide strong sovereignty over how this data is used, while storing less sensitive data with the operators to enable more convenient processing for analysis and allow larger logbook sizes, even with weak user hardware. Alternatively, some of the data could be duplicated on the user device to enable real-time decision-making during interactions, while ensuring the data's availability (and potentially larger logbook sizes) through operator storage.

Of course, designing such a combined system comes with many new challenges compared to the two systems presented in this thesis.

Bibliography

[AB09] Elli Androulaki and Steven M. Bellovin. "An Anonymous Credit Card System". In: *TrustBus 2009*. Ed. by Simone Fischer-Hübner, Costas Lambrinoudakis, and Günther Pernul. Vol. 5695. LNCS. Linz, Austria: Springer Berlin Heidelberg, Germany, Sept. 2009, pp. 42–51. DOI: 10.1007/978-3-642-03748-1_5.

[Abe+11] Masayuki Abe, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. "Optimal Structure-Preserving Signatures in Asymmetric Bilinear Groups". In: *CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. LNCS. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2011, pp. 649–666. DOI: 10.1007/978-3-642-22792-9_37.

[Abe+15] Masayuki Abe, Markulf Kohlweiss, Miyako Ohkubo, and Mehdi Tibouchi. "Fully Structure-Preserving Signatures and Shrinking Commitments". In: *EUROCRYPT 2015, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. LNCS. Sofia, Bulgaria: Springer Berlin Heidelberg, Germany, Apr. 2015, pp. 35–65. DOI: 10.1007/978-3-662-46803-6_2.

[Abe+16] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. "Structure-Preserving Signatures and Commitments to Group Elements". In: *Journal of Cryptology* 29.2 (Apr. 2016), pp. 363–421. DOI: 10.1007/s00145-014-9196-7.

[Add+22] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. "Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares". In: *SCN 22*. Ed. by Clemente Galdi and Stanislaw Jarecki. Vol. 13409. LNCS. Amalfi, Italy: Springer, Cham, Switzerland, Sept. 2022, pp. 516–539. DOI: 10.1007/978-3-031-14791-3_23.

[Adi08] Ben Adida. "Helios: Web-based Open-Audit Voting". In: *USENIX Security 2008*. Ed. by Paul C. van Oorschot. San Jose, CA, USA: USENIX Association, July 2008, pp. 335–348. URL: http://www.usenix.org/events/sec08/tech/full_papers/adida/adida.pdf.

[AG20] D. F. Aranha and C. P. L. Gouvêa. *RELIC is an Efficient LIbrary for Cryptography*. https://github.com/relic-toolkit/relic. 2020.

[Aim20] Aimia Coalition Loyalty UK Ltd. *The Nectar loyalty program*. Online Resource. https://www.nectar.com/. 2020.

[APR21] Diego F. Aranha, Elena Pagnin, and Francisco Rodríguez-Henríquez. "LOVE a Pairing". In: *LATINCRYPT 2021*. Ed. by Patrick Longa and Carla Ràfols. Vol. 12912. LNCS. Springer, Cham, Switzerland, Oct. 2021, pp. 320–340. DOI: 10.1007/978-3-030-88238-9_16.

[Ara+11] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio Cesar López-Hernández. "Faster Explicit Formulas for Computing Pairings over Ordinary Curves". In: *EUROCRYPT 2011*. Ed. by Kenneth G. Paterson. Vol. 6632. LNCS. Tallinn, Estonia: Springer Berlin Heidelberg, Germany, May 2011, pp. 48–68. DOI: 10.1007/978-3-642-20465-4_5.

[Arf+15] Ghada Arfaoui, Jean-François Lalande, Jacques Traoré, Nicolas Desmoulins, Pascal Berthomé, and Saïd Gharout. "A Practical Set-Membership Proof for Privacy-Preserving NFC Mobile Ticketing". In: *PoPETs* 2015.2 (Apr. 2015), pp. 25–45. DOI: 10.1515/popets-2015-0019.

[Ash+17] Gilad Asharov, Shai Halevi, Yehuda Lindell, and Tal Rabin. *Privacy-Preserving Search of Similar Patients in Genomic Data*. Cryptology ePrint Archive, Report 2017/144. 2017. URL: https://eprint.iacr.org/2017/144.

[Au+14] Man Ho Au, Joseph K. Liu, Junbin Fang, Zoe L. Jiang, Willy Susilo, and Jianying Zhou. "A New Payment System for Enhancing Location Privacy of Electric Vehicles". In: *IEEE Transactions on Vehicular Technology* 63.1 (2014), pp. 3–18. DOI: 10.1109/TVT.2013.2274288.

[Bac+12] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. "ObliviAd: Provably Secure and Practical Online Behavioral Advertising". In: *2012 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2012, pp. 257–271. DOI: 10.1109/SP.2012.25.

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. "Compact Multi-signatures for Smaller Blockchains". In: *ASIACRYPT 2018, Part II*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11273. LNCS. Brisbane, Queensland, Australia: Springer, Cham, Switzerland, Dec. 2018, pp. 435–464. DOI: 10.1007/978-3-030-03329-3_15.

[Bel+23] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. "ACORN: Input Validation for Secure Aggregation". In: *USENIX Security 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. Anaheim, CA, USA: USENIX Association, Aug. 2023, pp. 4805–4822. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/bell.

[Ber+23a] Robin Berger, Brandon Broadnax, Michael Klooß, Jeremias Mechler, Jörn Müller-Quade, Astrid Ottenhues, and Markus Raiber. "Composable Long-Term Security with Rewinding". In: *TCC 2023, Part IV*. Ed. by Guy N. Rothblum and Hoeteck Wee. Vol. 14372. LNCS. Taipei, Taiwan: Springer, Cham, Switzerland, Nov. 2023, pp. 510–541. DOI: 10.1007/978-3-031-48624-1_19.

[Ber+23b] Robin Berger, Brandon Broadnax, Michael Klooß, Jeremias Mechler, Jörn Müller-Quade, Astrid Ottenhues, and Markus Raiber. "Composable Long-Term Security with Rewinding". In: *TCC 2023, Part IV*. Ed. by Guy N. Rothblum and Hoeteck Wee. Vol. 14372. LNCS. Taipei, Taiwan: Springer, Cham, Switzerland, Nov. 2023, pp. 510–541. DOI: 10.1007/978-3-031-48624-1_19.

[Blö+19] Johannes Blömer, Jan Bobolz, Denis Diemert, and Fabian Eidens. "Updatable Anonymous Credentials and Applications to Incentive Systems". In: *ACM CCS 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. London, UK: ACM Press, Nov. 2019, pp. 1671–1685. DOI: 10.1145/3319535.3354223.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short Signatures from the Weil Pairing". In: *ASIACRYPT 2001*. Ed. by Colin Boyd. Vol. 2248. LNCS. Gold Coast, Australia: Springer Berlin Heidelberg, Germany, Dec. 2001, pp. 514–532. DOI: 10.1007/3-540-45682-1_30.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. "Short Signatures from the Weil Pairing". In: *Journal of Cryptology* 17.4 (Sept. 2004), pp. 297–319. DOI: 10.1007/s00145-004-0314-9.

[Blu81] Manuel Blum. "Coin Flipping by Telephone". In: *CRYPTO'81*. Ed. by Allen Gersho. Vol. ECE Report 82-04. Santa Barbara, CA, USA: U.C. Santa Barbara, Dept. of Elec. and Computer Eng., 1981, pp. 11–15.

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. "Pairing-Friendly Elliptic Curves of Prime Order". In: *SAC 2005*. Ed. by Bart Preneel and Stafford Tavares. Vol. 3897. LNCS. Kingston, Ontario, Canada: Springer Berlin Heidelberg, Germany, Aug. 2006, pp. 319–331. DOI: 10.1007/11693383_22.

[Bon+21] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. "Lightweight Techniques for Private Heavy Hitters". In: *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2021, pp. 762–776. DOI: 10.1109/SP40001.2021.00048.

[BR 18] Brazilian Federal Government. *Lei Geral de Proteção de Dados Pessoais (LGPD) – Law No. 13.709/2018*. Law. Official publication in the Diário Oficial da União. Aug. 14, 2018. URL: https://www.planalto.gov.br/ccivil_03/_Ato2015-2018/2018/Lei/L13709.htm (visited on 01/15/2026).

[Bra+23] Lennart Braun, Mahak Pancholi, Rahul Rachuri, and Mark Simkin. "Ramen: Souper Fast Three-Party Computation for RAM Programs". In: *ACM CCS 2023*. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. Copenhagen, Denmark: ACM Press, Nov. 2023, pp. 3284–3297. DOI: 10.1145/3576915.3623115.

[Bün+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020.

[CA 18] State of California. *California Consumer Privacy Act (CCPA) of 2018*. Statute. Effective 1 January 2020; amended by the California Privacy Rights Act (CPRA) in 2020. June 28, 2018. URL: https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375 (visited on 01/15/2026).

[Cam18] Kattankulathur Campus. "Credit card fraud detection using machine learning models and collating machine learning models". In: *International Journal of Pure and Applied Mathematics* 118.20 (2018), pp. 825–838.

[Can+18] Rémi Canillas, Rania Talbi, Sara Bouchenak, Omar Hasan, Lionel Brunie, and Laurent Sarrat. "Exploratory Study of Privacy Preserving Fraud Detection". In: *19th International Middleware Conference 2018*. ACM, 2018, pp. 25–31. DOI: 10.1145/3284028.3284032. URL: https://doi.org/10.1145/3284028.3284032.

[Can00] Ran Canetti. *Universally Composable Security: A New Paradigm for Cryptographic Protocols*. Cryptology ePrint Archive, Report 2000/067. 2000. URL: https://eprint.iacr.org/2000/067.

[Can01] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. Las Vegas, NV, USA: IEEE Computer Society Press, Oct. 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[Can20] Ran Canetti. "Universally Composable Security". In: *J. ACM* 67.5 (Sept. 2020). DOI: 10.1145/3402457. URL: https://doi.org/10.1145/3402457.

[CB17] Henry Corrigan-Gibbs and Dan Boneh. "Prio: Private, Robust, and Scalable Computation of Aggregate Statistics". In: *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. Ed. by Aditya Akella and Jon Howell. USENIX Association, 2017, pp. 259–282. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs.

[Cet+17] Gizem S Cetin, Hao Chen, Kim Laine, Kristin Lauter, Peter Rindal, and Yuhou Xia. *Private Queries on Encrypted Genomic Data*. Cryptology ePrint Archive, Report 2017/207. 2017. URL: https://eprint.iacr.org/2017/207.

[CF01] Ran Canetti and Marc Fischlin. "Universally Composable Commitments". In: *CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. LNCS. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2001, pp. 19–40. DOI: 10.1007/3-540-44647-8_2.

[Che+17] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. *Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning*. 2017.

[Che+24] Ying Chen, Debiao He, Zijian Bao, Min Luo, and Kim-Kwang Raymond Choo. "A Post-Quantum Privacy-Preserving Payment Protocol in Vehicle to Grid Networks". In: *IEEE Transactions on Intelligent Vehicles* (2024), pp. 1–13. DOI: 10.1109/TIV.2024.3374724.

[Dam+12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. "Multiparty Computation from Somewhat Homomorphic Encryption". In: *CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. LNCS. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2012, pp. 643–662. DOI: 10.1007/978-3-642-32009-5_38.

[Dav+23] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. "Verifiable Distributed Aggregation Functions". In: *PoPETs* 2023.4 (Oct. 2023), pp. 578–592. DOI: 10.56553/popets-2023-0126.

[DT08] Ivan Damgard and Rune Thorbek. *Efficient Conversion of Secret-shared Values Between Different Fields*. Cryptology ePrint Archive, Report 2008/221. 2008. URL: https://eprint.iacr.org/2008/221.

[DZ16] Tamara M. Dugan and Xukai Zou. "A Survey of Secure Multiparty Computation Protocols for Privacy Preserving Genetic Tests". In: *Proceedings of the First IEEE International Conference on Connected Health: Applications, Systems and Engineering Technologies, CHASE 2016, Washington, DC, USA, June 27-29, 2016*. IEEE Computer Society, 2016, pp. 173–182. DOI: 10.1109/CHASE.2016.71. URL: https://doi.org/10.1109/CHASE.2016.71.

[EG14] Alex Escala and Jens Groth. "Fine-Tuning Groth-Sahai Proofs". In: *PKC 2014*. Ed. by Hugo Krawczyk. Vol. 8383. LNCS. Buenos Aires, Argentina: Springer Berlin Heidelberg, Germany, Mar. 2014, pp. 630–649. DOI: 10.1007/978-3-642-54631-0_36.

[EPC16] European Parliament and Council. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation)*. Regulation. Official Journal of the European Union L 119, 4 May 2016, pp. 1-88. EUR-L, May 4, 2016. URL: http://data.europa.eu/eli/reg/2016/679/2016-05-04 (visited on 01/15/2026).

[EPK14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. "RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response". In: *ACM CCS 2014*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. Scottsdale, AZ, USA: ACM Press, Nov. 2014, pp. 1054–1067. DOI: 10.1145/2660267.2660348.

[Eur17] European Central Bank (ECB). *Average number of cash and card transactions per person per day in the Euro Area in 2016, by country*. Website. https://www.statista.com/statistics/893459/average-number-of-transactions-per-person-per-day-by-method/, last visited 2021-11-18. Nov. 2017.

[Eur18] European Central Bank (ECB). *Fifth Report on card fraud, September 2018*. Website. https://www.ecb.europa.eu/pub/cardfraud/html/ecb.cardfraudreport201809.en.html, last visited 2020-04-27. Sept. 2018.

[Fal+21a] Sebastian H. Faller, Pascal Baumer, Michael Klooß, Alexander Koch, Astrid Ottenhues, and Markus Raiber. "Black-Box Accumulation Based on Lattices". In: *18th IMA International Conference on Cryptography and Coding*. Ed. by Maura B. Paterson. Vol. 13129. LNCS. Virtual Event: Springer, Cham, Switzerland, Dec. 2021, pp. 220–246. DOI: 10.1007/978-3-030-92641-0_11.

[Fal+21b] Sebastian H. Faller, Pascal Baumer, Michael Klooß, Alexander Koch, Astrid Ottenhues, and Markus Raiber. "Black-Box Accumulation Based on Lattices". In: *18th IMA International Conference on Cryptography and Coding*. Ed. by Maura B. Paterson. Vol. 13129. LNCS. Virtual Event: Springer, Cham, Switzerland, Dec. 2021, pp. 220–246. DOI: 10.1007/978-3-030-92641-0_11.

[Fal+23] Brett Hemenway Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. "GigaDORAM: Breaking the Billion Address Barrier". In: *USENIX Security 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. Anaheim, CA, USA: USENIX Association, Aug. 2023, pp. 3871–3888. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/falk.

[Fau+25a] Dennis Faut, Valerie Fetzer, Jörn Müller-Quade, Markus Raiber, and Andy Rupp. *POBA: Privacy-Preserving Operator-Side Bookkeeping and Analytics*. Cryptology ePrint Archive, Report 2025/801. 2025. URL: https://eprint.iacr.org/2025/801.

[Fau+25b] Dennis Faut, Valerie Fetzer, Jörn Müller-Quade, Markus Raiber, and Andy Rupp. "POBA: Privacy-Preserving Operator-Side Bookkeeping and Analytics". In: *IACR Commun. Cryptol.* 2.2 (2025), p. 7. DOI: 10.62056/AV11ZO-3Y.

[Fau+25c] Dennis Faut, Valerie Fetzer, Jörn Müller-Quade, Markus Raiber, and Andy Rupp. "POBA: Privacy-Preserving Operator-Side Bookkeeping and Analytics". In: *IACR Commun. Cryptol.* 2.2 (2025), p. 7. DOI: 10.62056/AV11ZO-3Y.

[Fet+21] Valerie Fetzer, Marcel Keller, Sven Maier, Markus Raiber, Andy Rupp, and Rebecca Schwerdt. *PUBA: Privacy-Preserving User-Data Bookkeeping and Analytics*. Cryptology ePrint Archive, Report 2021/1683. 2021. URL: https://eprint.iacr.org/2021/1683.

[Fet+22a] Valerie Fetzer, Marcel Keller, Sven Maier, Markus Raiber, Andy Rupp, and Rebecca Schwerdt. "PUBA: Privacy-Preserving User-Data Bookkeeping and Analytics". In: *PoPETs* 2022.2 (Apr. 2022), pp. 447–516. DOI: 10.2478/popets-2022-0054.

[Fet+22b] Valerie Fetzer, Marcel Keller, Sven Maier, Markus Raiber, Andy Rupp, and Rebecca Schwerdt. "PUBA: Privacy-Preserving User-Data Bookkeeping and Analytics". In: *PoPETs* 2022.2 (Apr. 2022), pp. 447–516. DOI: 10.2478/popets-2022-0054.

[Fet+23a] Valerie Fetzer, Michael Klooß, Jörn Müller-Quade, Markus Raiber, and Andy Rupp. "Universally Composable Auditable Surveillance". In: *ASIACRYPT 2023, Part II*. Ed. by Jian Guo and Ron Steinfeld. Vol. 14439. LNCS. Guangzhou, China: Springer, Singapore, Singapore, Dec. 2023, pp. 453–487. DOI: 10.1007/978-981-99-8724-5_14.

[Fet+23b] Valerie Fetzer, Michael Klooß, Jörn Müller-Quade, Markus Raiber, and Andy Rupp. "Universally Composable Auditable Surveillance". In: *ASIACRYPT 2023, Part II*. Ed. by Jian Guo and Ron Steinfeld. Vol. 14439. LNCS. Guangzhou, China: Springer, Singapore, Singapore, Dec. 2023, pp. 453–487. DOI: 10.1007/978-981-99-8724-5_14.

[FPE16] Giulia C. Fanti, Vasyl Pihur, and Úlfar Erlingsson. "Building a RAPPOR with the Unknown: Privacy-Preserving Learning of Associations and Data Dictionaries". In: *PoPETs* 2016.3 (July 2016), pp. 41–61. DOI: 10.1515/popets-2016-0015.

[Gao+20] Yansong Gao, Bao Gia Doan, Zhi Zhang, Siqi Ma, Jiliang Zhang, Anmin Fu, Surya Nepal, and Hyoungshick Kim. *Backdoor Attacks and Countermeasures on Deep Learning: A Comprehensive Review*. 2020.

[GCF11] Saikat Guha, Bin Cheng, and Paul Francis. "Privad: Practical Privacy in Online Advertising". In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. Ed. by David G. Andersen and Sylvia Ratnasamy. USENIX Association, 2011. URL: https://www.usenix.org/conference/nsdi11/privad-practical-privacy-online-advertising.

[GLM16] Matthew Green, Watson Ladd, and Ian Miers. "A Protocol for Privately Reporting Ad Impressions at Scale". In: *ACM CCS 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. Vienna, Austria: ACM Press, Oct. 2016, pp. 1591–1601. DOI: 10.1145/2976749.2978407.

[GM82] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information". In: *14th ACM STOC*. San Francisco, CA, USA: ACM Press, May 1982, pp. 365–377. DOI: 10.1145/800070.802212.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *19th ACM STOC*. Ed. by Alfred Aho. New York City, NY, USA: ACM Press, May 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004. DOI: 10.1017/CBO9780511721656. URL: http://www.wisdom.weizmann.ac.il/%5C%7Eoded/foc-vol2.html.

[Gol87] Oded Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs". In: *19th ACM STOC*. Ed. by Alfred Aho. New York City, NY, USA: ACM Press, May 1987, pp. 182–194. DOI: 10.1145/28395.28416.

[Gro06] Jens Groth. "Simulation-Sound NIZK Proofs for a Practical Language and Constant Size Group Signatures". In: *ASIACRYPT 2006*. Ed. by Xuejia Lai and Kefei Chen. Vol. 4284. LNCS. Shanghai, China: Springer Berlin Heidelberg, Germany, Dec. 2006, pp. 444–459. DOI: 10.1007/11935230_29.

[GS08] Jens Groth and Amit Sahai. "Efficient Non-interactive Proof Systems for Bilinear Groups". In: *EUROCRYPT 2008*. Ed. by Nigel P. Smart. Vol. 4965. LNCS. Istanbul, Turkey: Springer Berlin Heidelberg, Germany, Apr. 2008, pp. 415–432. DOI: 10.1007/978-3-540-78967-3_24.

[Gud14] Ivan Gudymenko. "A Privacy-Preserving E-Ticketing System for Public Transportation Supporting Fine-Granular Billing and Local Validation". In: *Proceedings of the 7th International Conference on Security of Information and Networks*. SIN '14. Glasgow, Scotland, UK: Association for Computing Machinery, 2014, pp. 101–108. DOI: 10.1145/2659651.2659706. URL: https://doi.org/10.1145/2659651.2659706.

[Hal+26] Marius Haller, Marc Leinweber, Tilo Spannagel, Markus Raiber, and Hannes Hartenstein. "Performance and Security of TEE-Based Threshold Cryptography". In: *Proceedings of the 9th Workshop on System Software for Trusted Execution, SysTEX 2026*. Ed. by Edlira Dushku and Adrien Ghosn. ACM, 2026, pp. 42–49. DOI: 10.1145/3805690.3805725.

[Han+21] Jinguang Han, Liqun Chen, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. "Privacy-Preserving Electronic Ticket Scheme with Attribute-Based Credentials". In: *IEEE Transactions on Dependable and Secure Computing* 18.4 (2021), pp. 1836–1849. DOI: 10.1109/TDSC.2019.2940946.

[Har+17] Gunnar Hartung, Max Hoffmann, Matthias Nagel, and Andy Rupp. "BBA+: Improving the Security and Applicability of Privacy-Preserving Point Collection". In: *ACM CCS 2017*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 1925–1942. DOI: 10.1145/3133956.3134071.

[Hey+06] Thomas S. Heydt-Benjamin, Hee-Jin Chae, Benessa Defend, and Kevin Fu. "Privacy for Public Transportation". In: *PET 2006*. Ed. by George Danezis and Philippe Golle. Vol. 4258. LNCS. Cambridge, UK: Springer Berlin Heidelberg, Germany, June 2006, pp. 1–19. DOI: 10.1007/11957454_1.

[HJ12] Dennis Hofheinz and Tibor Jager. "Tightly Secure Signatures and Public-Key Encryption". In: *CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. LNCS. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2012, pp. 590–607. DOI: 10.1007/978-3-642-32009-5_35.

[Hof+20a] Max Hoffmann, Michael Klooß, Markus Raiber, and Andy Rupp. "Black-Box Wallets: Fast Anonymous Two-Way Payments for Constrained Devices". In: *PoPETs* 2020.1 (Jan. 2020), pp. 165–194. DOI: 10.2478/popets-2020-0010.

[Hof+20b] Max Hoffmann, Michael Klooß, Markus Raiber, and Andy Rupp. "Black-Box Wallets: Fast Anonymous Two-Way Payments for Constrained Devices". In: *PoPETs* 2020.1 (Jan. 2020), pp. 165–194. DOI: 10.2478/popets-2020-0010.

[IMS20] Fayaz Itoo, Meenakshi, and Satwinder Singh. "Comparison and analysis of logistic regression, Naïve Bayes and KNN machine learning algorithms for credit card fraud detection". In: *International Journal of Information Technology* (2020), pp. 1–9. DOI: 10.1007/s41870-020-00430-yj. URL: https://doi.org/10.1007/s41870-020-00430-y.

[Ind23] Government of India. *Digital Personal Data Protection Act, 2023*. Act. Published in the Gazette of India, Part II, Section 3. Aug. 3, 2023. URL: https://www.meity.gov.in/static/uploads/2024/06/2bf1f0e9f04e6fb4f8fef35e82c42aa5.pdf (visited on 01/15/2026).

[JJQ07] Oliver Jorns, Oliver Jung, and Gerald Quirchmayr. "A Privacy Enhancing Service Architecture for Ticket-based Mobile Applications". In: *The Second International Conference on Availability, Reliability and Security (ARES'07)*. 2007, pp. 139–146. DOI: 10.1109/ARES.2007.16.

[JR16] Tibor Jager and Andy Rupp. "Black-Box Accumulation: Collecting Incentives in a Privacy-Preserving Way". In: *PoPETs* 2016.3 (July 2016), pp. 62–82. DOI: 10.1515/popets-2016-0016.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "GAZELLE: A Low Latency Framework for Secure Neural Network Inference". In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. Baltimore, MD, USA: USENIX Association, Aug. 2018, pp. 1651–1669. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar.

[Kel20] Marcel Keller. "MP-SPDZ: A Versatile Framework for Multi-Party Computation". In: *ACM CCS 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. Virtual Event, USA: ACM Press, Nov. 2020, pp. 1575–1590. DOI: 10.1145/3372297.3417872.

[KKS12] Vladimir Kolesnikov, Ranjit Kumaresan, and Abdullatif Shikfa. "Efficient Verification of Input Consistency in Server-Assisted Secure Function Evaluation". In: *CANS 12*. Ed. by Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis. Vol. 7712. LNCS. Darmstadt, Germany: Springer Berlin Heidelberg, Germany, Dec. 2012, pp. 201–217. DOI: 10.1007/978-3-642-35404-5_16.

[KLG13] Florian Kerschbaum, Hoon Wei Lim, and Ivan Gudymenko. "Privacy-Preserving Billing for e-Ticketing Systems in Public Transportation". In: *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*. WPES '13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 143–154. DOI: 10.1145/2517840.2517848. (Visited on 02/20/2024).

[Kob87] Neal Koblitz. "Elliptic Curve Cryptosystems". In: *Mathematics of Computation* 48.177 (Jan. 1987), p. 203. DOI: 10.2307/2007884. URL: http://dx.doi.org/10.2307/2007884.

[Kon+20] Dániel Kondor, Behrooz Hashemian, Yves-Alexandre de Montjoye, and Carlo Ratti. "Towards Matching User Mobility Traces in Large-Scale Datasets". In: *IEEE Transactions on Big Data* 6.4 (2020), pp. 714–726. DOI: 10.1109/TBDATA.2018.2871693. URL: https://ieeexplore.ieee.org/document/8470173.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer". In: *ACM CCS 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. Vienna, Austria: ACM Press, Oct. 2016, pp. 830–842. DOI: 10.1145/2976749.2978357.

[KS14] Marcel Keller and Peter Scholl. "Efficient, Oblivious Data Structures for MPC". In: *ASIACRYPT 2014, Part II*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8874. LNCS. Kaoshiung, Taiwan, R.O.C.: Springer Berlin Heidelberg, Germany, Dec. 2014, pp. 506–525. DOI: 10.1007/978-3-662-45608-8_27.

[Lob20] Loblaw Companies. *PC Optimum loyalty program*. Online Resource. https://www.pcoptimum.ca. 2020.

[LP09] Yehuda Lindell and Benny Pinkas. "Secure Multiparty Computation for Privacy-Preserving Data Mining". In: *Journal of Privacy and Confidentiality* 1.1 (Apr. 2009), pp. 59–98. DOI: 10.29012/jpc.v1i1.566.

[MDD16] Luca Melis, George Danezis, and Emiliano De Cristofaro. "Efficient Private Statistics with Succinct Sketches". In: *NDSS 2016*. San Diego, CA, USA: The Internet Society, Feb. 2016. DOI: 10.14722/ndss.2016.23175.

[Mil+15] Milica Milutinovic, Koen Decroix, Vincent Naessens, and Bart De Decker. "Privacy-Preserving Public Transport Ticketing System". In: *Data and Applications Security and Privacy XXIX*. Ed. by Pierangela Samarati. Cham: Springer International Publishing, 2015, pp. 135–150. DOI: 10.1007/978-3-319-20810-7_9.

[Mit+24] Aikaterini Mitrokotsa, Sayantan Mukherjee, Mahdi Sedaghat, Daniel Slamanig, and Jenit Tomy. "Threshold Structure-Preserving Signatures: Strong and Adaptive Security Under Standard Assumptions". In: *PKC 2024, Part I*. Ed. by Qiang Tang and Vanessa Teague. Vol. 14601. LNCS. Sydney, NSW, Australia: Springer, Cham, Switzerland, Apr. 2024, pp. 163–195. DOI: 10.1007/978-3-031-57718-5_6.

[MR18] Payman Mohassel and Peter Rindal. "ABY^3: A Mixed Protocol Framework for Machine Learning". In: *ACM CCS 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 35–52. DOI: 10.1145/3243734.3243760.

[MST24] Dimitris Mouris, Pratik Sarkar, and Nektarios Georgios Tsoutsos. "PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries". In: *PoPETs* 2024.3 (July 2024), pp. 4–24. DOI: 10.56553/popets-2024-0064.

[PAY20] PAYBACK GmbH. *The Payback loyalty program*. Online Resource. https://www.payback.net/. 2020.

[PC15] European Parliament and Council. *PSD2: 2nd Payment Services Directive (2015/2366)*. https://eur-lex.europa.eu. 2015. URL: http://data.europa.eu/eli/dir/2015/2366/2015-12-23.

[RD11] Alfredo Rial and George Danezis. "Privacy-preserving smart metering". In: *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society, WPES 2011, Chicago, IL, USA, October 17, 2011*. Ed. by Yan Chen and Jaideep Vaidya. ACM, 2011, pp. 49–60. DOI: 10.1145/2046556.2046564. URL: https://doi.org/10.1145/2046556.2046564.

[RDK18] Alfredo Rial, George Danezis, and Markulf Kohlweiss. "Privacy-preserving smart metering revisited". In: *Int. J. Inf. Sec.* 17.1 (2018), pp. 1–31. DOI: 10.1007/S10207-016-0355-8. URL: https://doi.org/10.1007/s10207-016-0355-8.

[Rup+13] Andy Rupp, Gesine Hinterwälder, Foteini Baldimtsi, and Christof Paar. "P4R: Privacy-Preserving Pre-Payments with Refunds for Transportation Systems". In: *FC 2013*. Ed. by Ahmad-Reza Sadeghi. Vol. 7859. LNCS. Okinawa, Japan: Springer Berlin Heidelberg, Germany, Apr. 2013, pp. 205–212. DOI: 10.1007/978-3-642-39884-1_17.

[Sch+19] Rebecca Schwerdt, Matthias Nagel, Valerie Fetzer, Tobias Gräf, and Andy Rupp. "P6V2G: a privacy-preserving V2G scheme for two-way payments and reputation". In: *Energy Inform.* 2.S1 (2019). DOI: 10.1186/S42162-019-0075-1. URL: https://doi.org/10.1186/s42162-019-0075-1.

[Sen+05] Andrew W. Senior, Sharath Pankanti, Arun Hampapur, Lisa M. Brown, Ying-li Tian, Ahmet Ekin, Jonathan H. Connell, Chiao-Fe Shu, and Max Lu. "Enabling Video Privacy through Computer Vision". In: *IEEE Secur. Priv.* 3.3 (2005), pp. 50–57. DOI: 10.1109/MSP.2005.65. URL: https://doi.org/10.1109/MSP.2005.65.

[SG02] Victor Shoup and Rosario Gennaro. "Securing Threshold Cryptosystems against Chosen Ciphertext Attack". In: *Journal of Cryptology* 15.2 (Mar. 2002), pp. 75–96. DOI: 10.1007/s00145-001-0020-9.

[Ste+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. "Path ORAM: an extremely simple oblivious RAM protocol". In: *ACM CCS 2013*. Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung. Berlin, Germany: ACM Press, Nov. 2013, pp. 299–310. DOI: 10.1145/2508859.2516660.

[Ste+21a] Oliver Stengele, Markus Raiber, Jörn Müller-Quade, and Hannes Hartenstein. "ETHTID: Deployable Threshold Information Disclosure on Ethereum". In: *Third International Conference on Blockchain Computing and Applications, BCCA 2021, Tartu, Estonia, November 15-17, 2021*. IEEE, 2021, pp. 127–134. DOI: 10.1109/BCCA53669.2021.9657019.

[Ste+21b] Oliver Stengele, Markus Raiber, Jörn Müller-Quade, and Hannes Hartenstein. "ETHTID: Deployable Threshold Information Disclosure on Ethereum". In: *Third International Conference on Blockchain Computing and Applications, BCCA 2021, Tartu, Estonia, November 15-17, 2021*. IEEE, 2021, pp. 127–134. DOI: 10.1109/BCCA53669.2021.9657019.

[Tou+10] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. "Adnostic: Privacy Preserving Targeted Advertising". In: *NDSS 2010*. San Diego, CA, USA: The Internet Society, Feb. 2010.

[Tra+16] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Stealing Machine Learning Models via Prediction APIs". In: *USENIX Security 2016*. Ed. by Thorsten Holz and Stefan Savage. Austin, TX, USA: USENIX Association, Aug. 2016, pp. 601–618. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer.

[Val76] Leslie G. Valiant. "Universal Circuits (Preliminary Report)". In: *Proceedings of the 8th Annual ACM Symposium on Theory of Computing*. Ed. by Ashok K. Chandra, Detlef Wotschke, Emily P. Friedman, and Michael A. Harrison. Hershey, Pennsylvania, USA: ACM, 1976, pp. 196–203. DOI: 10.1145/800113.803649.

[Viv+10] Arnau Vives-Guasch, Jordi Castellà-Roca, M. Magdalena Payeras-Capella, and Macià Mut-Puigserver. "An electronic and secure automatic fare collection system with revocable anonymity for users". In: *Proceedings of the 8th International Conference on Advances in Mobile Computing and Multimedia*. MoMM '10. Paris, France: Association for Computing Machinery, 2010, pp. 387–392. DOI: 10.1145/1971519.1971585. URL: https://doi.org/10.1145/1971519.1971585.

[Wan+19] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. "Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 707–723. DOI: 10.1109/SP.2019.00031. URL: https://doi.org/10.1109/SP.2019.00031.

[Wan+22] Zhiguo Wan, Tong Zhang, Weizhuang Liu, Mingqiang Wang, and Liehuang Zhu. "Decentralized Privacy-Preserving Fair Exchange Scheme for V2G Based on Blockchain". In: *IEEE Transactions on Dependable and Secure Computing* 19.4 (2022), pp. 2442–2456. DOI: 10.1109/TDSC.2021.3059345.

[Win20] Wayne Winston. "How Does Target Know If You're Pregnant?" In: *Analytics Stories*. John Wiley & Sons, Ltd, 2020. Chap. 28, pp. 219–224. DOI: 10.1002/9781119646068.ch28. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119646068.ch28.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. *EMP-toolkit: Efficient MultiParty computation toolkit*. https://github.com/emp-toolkit. 2016.

[WR10] Thomas Winkler and Bernhard Rinner. "A systematic approach towards user-centric privacy and security for smart camera networks". In: *2010 Fourth ACM/IEEE International Conference on Distributed Smart Cameras, Atlanta, GA, USA - August 31 - September 4, 2010*. Ed. by Marilyn Wolf, Gian Luca Foresti, and Horst Bischof. ACM, 2010, pp. 133–141. DOI: 10.1145/1865987.1866009. URL: https://doi.org/10.1145/1865987.1866009.

[Yu+18] Hyunwoo Yu, Jaemin Lim, Kiyeon Kim, and Suk-Bok Lee. “Pinto: Enabling Video Privacy for Commodity IoT Cameras”. In: *ACM CCS 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 1089–1101. DOI: 10.1145/3243734.3243830.

[Yue+24] Xiaohan Yue, Xue Bi, Haibo Yang, Shi Bai, and Yuan He. “PAP: A Privacy-Preserving Authentication Scheme With Anonymous Payment for V2G Networks”. In: *IEEE Transactions on Smart Grid* 15.6 (2024), pp. 6092–6111. DOI: 10.1109/TSG.2024.3435028.

[Zha+14] Tianyu Zhao, Chang Chen, Lingbo Wei, and Mengke Yu. “An Anonymous Payment System to Protect the Privacy of Electric Vehicles”. In: *2014 Sixth International Conference on Wireless Communications and Signal Processing (WCSP)*. Oct. 2014, pp. 1–6. DOI: 10.1109/WCSP.2014.6992208. (Visited on 04/13/2024).

List of Figures

List of Tables

List of Definitions

A. Appendix for Chapter 3

This chapter contains some additional information for PUBA, mainly the formal proof of security in Section A.3. Additionally, Section A.1 addresses some of the limitations of the scheme, while Section A.2 gives additional details on the explicit leakage in the ideal functionality $\mathcal{F}_{\text{PUBA}}$.

A.1. Discussion: On the Limitations of Our Scheme

This section addresses some limitations of PUBA, namely on how to verify privacy-friendliness of analysis functions, especially with respect to neural networks in Section A.1.1, of the consequences of aborts during protocol executions in Section A.1.2 and on the parameter choices of our benchmarks in Section A.1.1.

A.1.1. Verification by the TSA

A common problem of functions constructed using deep machine learning (such as neural networks) is a lack of transparency regarding their behavior. Our framework suffers from the same problem which even persists if we ignore function privacy for the operator; a user who has to compute a neural network on private data does not automatically know what the network computes and how the output is to be interpreted. A function that maps, say, purchases of a user to some abstract class of advertisements relevant for that user is hard to distinguish from one that maps purchases to an encoding that reflects the individual purchases on a fine-grained level.

In PUBA the problem is even harder as we additionally require function privacy for the operator; only a trusted signing authority is there to ensure that the operator only uses valid function parameters which provide a sufficient level of privacy for the user. The TSA has the same problems mentioned above: While it is straightforward to check whether a given machine learning model indeed classifies as specified for randomly chosen inputs, a sufficiently complex model can be used to hide *backdoors* [Che+17] in the form of special inputs provided by the operator which would break unlinkability and input privacy for any user. As it is highly unlikely that the TSA finds this backdoor by using random testing, the model behaves normally for all inputs chosen by the TSA with high probability and could even get certified. While we generally assume that the output of the function is a discrete set of much smaller size than the input space— as is the case for both applications

we propose—we do not restrict PUBA to this behavior; using arbitrary output for the operator requires special attention during the verification step.

While it is possible to implant a backdoor into the model given a sufficiently large output space and a sufficiently complex model, we stress that there are several different ways to detect—and even to remove, although at the cost of overall accuracy—a backdoor. A survey on the scenario itself alongside mechanisms to detect and remove a backdoor is given in [Gao+20].

The generality of PUBA allows the operator to create a model with a backdoor and submit it for the SignFP task, yet increasing progress in the field of backdoor detection [Gao+20; Wan+19] given only the final model makes it unlikely that these function parameters will get a certificate. We hence require the TSA to perform a number of such tests in order for the verification mechanism to be sufficiently daunting for an operator that tries to use backdoors.

A.1.2. The Case of Aborts

Aborts are a common problem in MPC: If a party loses connection during a computation or refuses to answer entirely then the computation cannot be finished. This is also modeled into most security frameworks. For example, in the asynchronous UC model we model our protocol in, the entire communication is managed by the adversary; parties can ask the adversary to transfer a given message to an other party but the adversary is free to change any part of the message. Using authenticated channels removes the adversaries capability to *change* the message and secure channels additionally take the adversaries ability to *read* the message. Yet even with these precautions the adversary is still able to *drop* messages at will.

For normal computations an abort only means that the parties do not get any output. But for PUBA this means that the user is at worst left with no valid logbook if the abort happened after the old logbook has been invalidated at the start of a task but before the new one has been created and sent to the user. An additional case to consider is if the abort happens during an Analytics task. This leaves the user incapable of ever outsourcing data again, as the linking number will never be reset.

Yet we stress that dealing with aborts is straightforward and could easily be incorporated into the protocol, albeit at the cost of a longer functionality and protocol description and a much more complicated security proof. But for completeness reasons we sketch here how the protocol can be secured against aborts.

In total there are four tasks where the user is directly involved and where aborting in between means that there is no logbook that the user can use and one task where an abort implies that the user cannot outsource anymore. The four tasks where the user is directly involved in, namely UReg, Bookkeeping, Outsource and Update, we have to ensure that the mechanism cannot be abused to let a malicious user obtain two different logbooks. Thus we require that the same messages that were sent before the abort will be sent in the

next interaction again to ensure that the reconstructed logbook will end up with the same serial number. So essentially the reconstruction only finishes the protocol using the state both parties had during the abort.

The situation only becomes complicated if an abort occurs during an Analytics task. Without a reconstruction mechanism the user would be unable to outsource ever again, as resetting the linking number *lin* to **0** is only possible in the Update task which requires a completed Analytics task. However, a slight modification suffices to deal with this case:

If an abort occurs during an Analytics task then this abort only matters if no output has been provided to the operator as parties generally get notified of the abort.[1] Hence the operator is aware that the computation involving data from a given linking number *lin* has failed. The reconstruction task basically consists of the update task but with all three manipulation vectors corresponding to $\perp$. That is, the permutation α is the identity, the direct update $\mathbf{s}$ is $\perp$ everywhere, and the additive increment is $\mathbf{a} = 0$. This resets the linking number *lin* stored inside the user's logbook to **0** and thus enables future Outsource tasks for that user.

Note that this reconstruction mechanism can be used to restore a broken logbook; yet until the reconstruction has been performed the user is locked from any further interactions, with the exception of aborts during Analytics, where the user can still perform Bookkeeping tasks.

A.1.3. On the Expressiveness of our Application Benchmark

We are not domain experts in fraud detection for mobile payments, and real-world parameters and implementation details for use cases like fraud detection are not easily obtainable. The simplified instantiations we benchmark in Section 3.6.4 should therefore be viewed as an educated guess how real-world systems could be parameterized. Some explanation for our parameter choices: On average, we have 0.3-0.8 credit card transactions per day per person in Europe [Eur17]. So 20 transactions per logbook (user history-size of 100) would cover about a month, while 200 transactions per logbook (user history-size of 1000) would cover at least 8 months. Older transactions are also implicitly taken into account by making the new risk level depend on the old risk level (and the new transactions). Moreover, Logistic Regression seems to be a reasonable method for credit card fraud detection (e.g., see [Cam18; IMS20]), therefore we used that for our analysis. The simple fraud detection mechanism we implemented is most likely simpler than mechanisms used in practice. One could of course extend the simple fraud detection we implemented with additional rules. Some ideas on how the simple fraud detection mechanism could be extended are:

1. Look at the location and time difference between transactions and deny them if it is not physically possible to travel the distance in that amount of time.

[1] If the abort is happening in the real world, then we can assume standard techniques such as timeouts can be used to determine that the message will likely never arrive.

UReg	Bookkeeping	Outsource	Analytics	Update
• Output $\mathtt{out}_O$ of Δ • pid of the user • Index of the used FP	• Output $(\alpha, \mathbf{s}, \mathtt{out}_O)$ of Δ • Index of the used FP	• Nothing	• Output $((\alpha_z, \mathbf{s}_z)_z, \mathtt{out}_O)$ of Δ • Index of the used values from Outsource • Index of the used FP	• Outputs $(\alpha, \mathbf{s})$ of Δ • Subsession identifier of the Outsource task

Table A.1.: Leakage of each task involving a user.

2. If the transaction amount is a lot higher than the average transaction amount, the number of remaining transactions until the next calculation gets decreased by more than one.
3. Implement daily limits: a limit on the number of transactions that are allowed per day or a limit on the total transaction value that is allowed per day or both.

It would be valuable to study how well our system models fraud detection systems used in practice, but that would be a research line of its own.

A.2. Leakage of the Tasks

We claim in Section 3.2.5 that we ensure unlinkability: the operator can identify the user in the task UReg and link the triplet of consecutive executions of Outsource, Analytics and Update to some anonymous user, but other than that any two tasks could have been performed by any two users inside the set of successfully registered users. We stress that in real applications the communication structure could still be used to link the same user to different tasks; yet other than that, the individual task executions leak no information to connect any two tasks to the same user outside the computation results. The leakage is listed in Table A.1.

In *UReg*, the user is de-anonymized. This is required to ensure that no user registers twice. In addition, the operator learns the output $\mathtt{out}_O$ of the function Δ alongside the used function parameters; this output is privacy preserving due to the requirements of the SignFP task.

In *Bookkeeping*, all that a corrupted operator learns are the used function parameters and the parts of the output of Δ we consider to be relevant for the operator, namely everything except for the additive increment. Again, ensuring unlinkability comes down to the SFP task.

In *Outsource*, the operator learns no identifying information from the behavior of the task itself but the tasks name leaks for which function an Analytics task has been scheduled.

In *Analytics*, the task execution already leaks the function specifier k through the inputs. As the number of parties required for each function is public knowledge and since it is known in which order the calls are executed, the operator knows which Outsource calls are relevant and used for this task. Thus linking the used data to their corresponding

Outsource call is trivially possible. Additionally, the operator once more obtains parts of the output from Δ which we require to be privacy-preserving.

In *Update*, the operator learns the subsession identifier of the Outsource call that caused the analytical computation. As this Outsource execution is already linked to an Analytics execution, the operator can link the entire triplet to the same user. Other than that, the operator learns no new information as the outputs for that user were already learned during the Analytics task execution.

A.3. Security

In this section, we prove the security of our system. That is, we show that the protocol Π_{PUBA} is *at least* as secure, as our ideal functionality $\mathcal{F}_{\text{PUBA}}$, without relying on a trusted party to execute $\mathcal{F}_{\text{PUBA}}$ on all parties inputs. To that end, we provide a simulator that simulates the protocol messages of honest parties without knowing the parties secret input, and prove that those simulated messages cannot be differentiated by any efficient environment $\mathcal{Z}$.

For technical reasons, we have to restrict our adversary to corrupting only *either* the proxy P, or the operator O. We split our simulator up in two parts. Section A.3.1 contains the simulator for all corruption scenarios related to the security of an honest user U, even in the presence of other malicious users. Section A.3.2 contains the simulator for all corruption scenarios regarding the security of an honest operator O. Combined, those two simulators cover all corruption scenarios, in which either P or O are honest.

This proof is taken verbatim from [Fet+22b] and uses slightly different notation for $\mathcal{F}_{\text{PUBA}}$ and Π_{PUBA}. Thus, we repeat both $\mathcal{F}_{\text{PUBA}}$ and Π_{PUBA} with their original notation here.

The Ideal Functionality $\mathcal{F}_{\text{PUBA}}$

$\mathcal{F}_{\text{PUBA}}$

This functionality facilitates user-centric privacy-preserving analytics. The function to be computed is specified by the global parameter Δ.

State:
The ideal function stores:
- Set $\mathcal{P}_{\text{User}}$ of registered users.
- $f_{\text{UH}}\colon pid_U \mapsto \mathcal{UH}$
- $f_{\text{OA}}\colon \mathcal{P}_{\text{User}} \to \{\text{true}, \text{false}\}$
- $f_{\text{OI}}\colon \{pid_P\} \times \{1, \ldots, K\}$ where $(pid_P,\ k)$ maps to a list $f_{\text{OI}}(pid_P,\ k)$ of entries $(ssid,\ pid_U,\ \mathcal{UH},\ input_U)$.
- $f_{\text{FP}}\colon \{\texttt{ureg} \cup \{\texttt{bookkeep}(K)\} \cup \{\texttt{analyt}K\}\} \times \mathbb{N} \to \{fp\}^*$, where different tasks are mapped to a list of function parameters fp.
- Partial mapping f_{UI} on $\mathcal{P}_{\text{User}}$. $pid_U \mapsto (ssid,\ \alpha,\ \mathbf{s},\ \mathbf{a},\ output_U)$.

Init:
Input O: (init).
Input $\mathcal{TSA}$: (init)
1. Respond to the other tasks.
Output O: (ok).
Output $\mathcal{TSA}$: (ok).

SignFP:
Input O: $(\texttt{signfp},\ fp,\ \texttt{task},\ input_O)$.
Input $\mathcal{TSA}$: $(\texttt{signfp},\ input_{\mathcal{TSA}})$.
1. Check[a] that $\Delta(\texttt{signfp},\ fp,\ \texttt{task},\ input_O,\ input_{\mathcal{TSA}}) = 1$.
2. Check[a] that fp have not been registered before for task $\texttt{task}$.
3. Let ℓ be $\min_\ell\colon f_{\text{FP}}(\texttt{task},\ \ell) = \bot$.
4. Set $f_{\text{FP}}[\texttt{task}, \ell] = fp$.
5. Leak $(\texttt{task},\ \ell)$ to the adversary.

Output O: $(\texttt{ok})$.
Output $\mathcal{TSA}$: $(\texttt{ok})$.

UReg:
Input O: $(\texttt{ureg}, fp, input_O)$.
1. Let[a] ℓ be the index for which $f_{\text{FP}}(\texttt{ureg},\ \ell) = fp$.
2. Leak ℓ to the adversary.

Input U: $(\texttt{ureg}, input_U)$.
1. Check[a] $pid_U \notin \mathcal{P}_{\text{User}}$
2. Append pid_U to $\mathcal{P}_{\text{User}}$.
3. $(\mathcal{UH}, output_U, output_O) \leftarrow \Delta(\texttt{ureg}, fp, input_U, input_O)$.
4. Append $(pid_U \mapsto \mathcal{UH})$ to f_{UH}.
5. Append $(pid_U \mapsto \text{false})$ to f_{OA}.

Output U: $(\mathcal{UH}, output_U)$.
Output O: $(output_O)$.

$\{\text{Bookkeeping}^{(k)}\}_{k=1}^{K}$:
Input O: $(\texttt{bookkeep}(k), fp, input_O)$.
1. Let[a] ℓ be $\min_\ell : f_{\text{FP}}(\texttt{bookkeep}(k),\ \ell) = fp$.
2. Leak ℓ to the adversary.

Input U: $(\texttt{bookkeep}(k), input_U)$.
1. Set[a] $\mathcal{UH} \coloneqq f_{\text{UH}}(pid_U)$.
2. $(\alpha, \mathbf{s}, \mathbf{a}, output_U, output_O) \leftarrow \Delta(\texttt{bookkeep}, fp, k, \mathcal{UH}, input_U, input_O)$.
3. $\mathcal{UH}' \leftarrow \alpha(\mathcal{UH})$.
4. $\mathcal{UH}'' \coloneqq (uh''_0, \ldots, uh''_{m-1})$ with $uh''_i \coloneqq \begin{cases} \mathbf{s}[i] & \text{for } \mathbf{s}[i] \neq \bot \\ \mathcal{UH}'[i] & \text{for } \mathbf{s}[i] = \bot \end{cases}$.
5. $f_{\text{UH}}(pid_U) \coloneqq \mathcal{UH}'' + \mathbf{a}$.
6. Set $f_{\text{UH}}(pid_U) \coloneqq \mathcal{UH}^{\text{new}}$.

Output U: $(\alpha, \mathbf{s}, \mathbf{a}, output_U)$.
Output O: $(output_O, \alpha, \mathbf{s})$

$\{\text{Outsource}^{(k)}\}_{k=1}^{K}$:
Input U: $(\texttt{outsourcek}, input_U)$.
Input P: $(\texttt{outsourcek})$.
Input O: $(\texttt{outsourcek})$.
1. Load current $ssid$.
2. If U and O are corrupted, append $(ssid, \bot, \bot, \bot)$ to $f_{\text{OI}}(pid_P, k)$ and skip all following steps.
3. Check[a] $f_{\text{OA}}(pid_U) \stackrel{?}{=} \text{false}$.
4. Set $f_{\text{OA}}(pid_U) \coloneqq \text{true}$.
5. Set[a] $\mathcal{UH} \coloneqq f_{\text{UH}}(pid_U)$.
6. Append $(ssid, pid_U, \mathcal{UH}, input_U)$ to $f_{\text{OI}}(pid_P, k)$.

Output U: $(\texttt{ok})$.
Output P: $(\texttt{ok})$.
Output O: $(\texttt{ok})$.

$\{\text{Analytics}^{(k)}\}_{k=1}^{K}$:
Input O: $(\texttt{analytk}, fp, input_O)$.
1. Let[a] ℓ be $\min_\ell : f_{\text{FP}}(\texttt{analytk},\ \ell) = fp$.
2. Leak ℓ to the adversary.

Input P: $(\texttt{analytk})$.
1. Load[a] and remove the first Z_k entries $\{(ssid_z, pid_z, \mathcal{UH}_z, input_z)\}_{z=1}^{Z_k}$ from $f_{\text{OI}}(pid_P, k)$.
2. If P is corrupted, ask for updated inputs $\{input_z \mid 1 \le z \le Z_k, pid_z \in \mathcal{P}_{\text{corr}}\}$.
3. If O is corrupted, ask for updated information $(\mathcal{UH}_z,\ input_z)$ for each $z \in \{1, \ldots, Z_k\}$ with $pid_z \notin \mathcal{P}_{\text{User}}$.
4. $\left(\{(\alpha_z, \mathbf{s}_z, \mathbf{a}_z, output_z)\}_{z=1}^{Z_k}, output_O\right) \leftarrow \Delta\left(\texttt{analyt}, fp, k, \{(\mathcal{UH}_z, input_z)\}_{z=1}^{Z}, input_O\right)$.
5. Leak $\{(z, \alpha_z, \mathbf{s}_z, \mathbf{a}_z, output_z) \mid 1 \le z \le Z_k, pid_z \in \mathcal{P}_{\text{corr}}\}$ to the adversary.
6. If O is corrupted, leak $\{(z,\ \alpha_z,\ \mathbf{s}_z,\ \mathbf{a}_z,\ output_z) \mid 1 \le z \le Z_k, pid_z \notin \mathcal{P}_{\text{User}}\}$ to the adversary.

7. For every z from 1 to Z_k with $pid_z \in \mathcal{P}_{\text{User}}$ set $f_{\text{UI}}(pid_z) \coloneqq (ssid_z, \alpha_z, \mathbf{s}_z, \mathbf{a}_z, output_z)$.

Output P: (ok).
Output O: $(\{\alpha_z, \mathbf{s}_z\}_{z=1}^{Z_k} output_O)$.

Update:
Input U: (update).
Input P: (update).
Input O: (update).

1. If U and O are corrupted, set $(\mathcal{UH}, output_U) = (\bot, \bot)$ and skip all following steps.
2. Set[a] $\mathcal{UH} \coloneqq f_{\text{UH}}(pid_U)$.
3. Set[a] $(ssid, \alpha, \mathbf{s}, \mathbf{a}, output_U) \coloneqq f_{\text{UI}}(pid_U)$.
4. If O is corrupted, leak $ssid$ to the adversary.
5. If U is honest and P is corrupted, leak $ssid$ to the adversary.
6. Remove $pid_U \mapsto (ssid, \alpha, \mathbf{s}, \mathbf{a}, output_U)$ from f_{UI}.
7. $\mathcal{UH}' \leftarrow \alpha(\mathcal{UH})$.
8. $\mathcal{UH}'' \coloneqq (uh''_0, \ldots, uh''_{m-1})$ with $uh''_i \coloneqq \begin{cases} \mathbf{s}[i] & \text{for } \mathbf{s}[i] \neq \bot \\ \mathcal{UH}'[i] & \text{for } \mathbf{s}[i] = \bot \end{cases}$.
9. Set $f_{\text{UH}}(pid_U) \coloneqq \mathcal{UH}'' + \mathbf{a}$.
10. Set $f_{\text{OA}}(pid_U) \coloneqq$ false.

Output U: $(\alpha,\ \mathbf{s},\ \mathbf{a},\ output_U)$.
Output P: (ok).
Output O: $(\alpha,\ \mathbf{s})$.

---

[a] If this fails, output $\bot$ and abort.

The Protocol Π_{PUBA}

Π_{PUBA}

This protocol facilitates user-centric privacy-preserving analytics.

State:
Each user U stores:
- User logbook Log
- User public key pk_U
- One-time pads o_α, $\mathsf{o}_\mathbf{s}$, $\mathsf{o}_\mathbf{a}$, o_{OUT} and o_{UNV}
- Verification key vk_O of the operator
- Verification key $\mathsf{vk}_{\mathcal{TSA}}$ of the trusted signing party

Each proxy P stores:
- Verification key vk_O of the operator
- Verification key $\mathsf{vk}_{\mathcal{TSA}}$ of the trusted signing party
- Mapping $f_{\text{OI}}^{(P)}$ on $\{1, \ldots, K\}$ that maps k to a list $f_{\text{OI}}^{(P)}(k)$ of entries $(lin,\ \langle \mathcal{UH} \rangle^P,\ \langle input_U \rangle^P,\ \langle \mathsf{o}_{\text{OUT}} \rangle^P,\ \langle \mathsf{o}_\alpha \rangle^P,\ \langle \mathsf{o}_\mathbf{s} \rangle^P,\ \langle \mathsf{o}_\mathbf{a} \rangle^P)$
- Partial mapping $f_{\text{UI}}^{(P)}$ on $\{lin\}$: $lin \mapsto (ct_\alpha,\ ct_\mathbf{s},\ ct_\mathbf{a},\ ct_{output_U})$

The operator O stores:
- Signature key pair $(\mathsf{vk}_O,\ \mathsf{sk}_O)$
- List L_{SER} of observed serial numbers
- Mapping $f_{\text{OI}}^{(O)}$ on $\{pid_P\} \times \{1, \ldots, K\}$ that maps $(pid_P,\ k)$ to a list $f_{\text{OI}}^{(O)}(pid_P,\ k)$ of entries $(lin,\ \langle \mathcal{UH} \rangle^O,\ \langle input_U \rangle^O,\ \langle \mathsf{o}_{\text{OUT}} \rangle^O,\ \langle \mathsf{o}_\alpha \rangle^O,\ \langle \mathsf{o}_\mathbf{s} \rangle^O,\ \langle \mathsf{o}_\mathbf{a} \rangle^O)$
- Partial mapping $f_{\text{UI}}^{(O)}$ on $\{lin\}$: $lin \mapsto (\alpha,\ \mathbf{s},\ com_\mathbf{a},\ ct_{output_U})$
- Mapping f_{FP} on fp that maps function parameters fp to a list $f_{\text{FP}}(fp)$ of tuples $(com_{fp},\ decom_{fp},\ \sigma_{fp})$.

The trusted signing party $\mathcal{TSA}$ stores:
- Signature key pair $(\mathsf{vk}_{\mathcal{TSA}},\ \mathsf{sk}_{\mathcal{TSA}})$

Init:
Input O: (init)
Input $\mathcal{TSA}$: (init)

O: Generate signature key pair:
- $(\mathsf{vk}_O, \mathsf{sk}_O) \leftarrow \text{SIG.Keygen}(\text{gp})$.
- Call[a] $\mathcal{F}_{\text{BB}}$ instance $sid_{\mathcal{F}_{\text{BB}}}$ with input $(\texttt{Register},\ pid_O,\ \mathsf{vk}_O)$.

$\mathcal{TSA}$: Generate signature key pair:
- $(\mathsf{vk}_{\mathcal{TSA}},\ \mathsf{sk}_{\mathcal{TSA}}) \leftarrow \text{SIG}'.\text{Keygen}(\text{gp})$.

– Call[a] $\mathcal{F}_{\text{BB}}$ instance $sid_{\mathcal{F}_{\text{BB}}}$ with input $(\texttt{Register}, pid_{\mathcal{TSA}}, \mathsf{vk}_{\mathcal{TSA}})$.
Output O: $(\texttt{ok})$
Output $\mathcal{TSA}$: $(\texttt{ok})$

SignFP:
Input O: Message $\texttt{signfp}$, FPs fp, task $\texttt{task}$, aux. input $input_O$
Input $\mathcal{TSA}$: Message $\texttt{signfp}$, aux. input $input_{\mathcal{TSA}}$.
O: $(com_{fp}, decom_{fp}) \leftarrow \mathsf{COM}'.\mathsf{commit}(fp)$.
$O \twoheadrightarrow \mathcal{TSA}$: $(fp, com_{fp}, decom_{fp}, \texttt{task}, input_O)$
$\mathcal{TSA}$: Check[a] that $\Delta(\texttt{signfp}, fp, \texttt{task}, input_{Operator}, input_{\mathcal{TSA}}) = 1$.
$\mathcal{TSA}$: Check[a] that $\mathsf{COM}'.\mathsf{open}(com_{fp}, decom_{fp}, fp) = 1$.
$\mathcal{TSA}$: $\sigma_{fp} \leftarrow \mathsf{SIG}'.\mathsf{Sign}(\mathsf{sk}_{\mathcal{TSA}}, (\texttt{task}, com_{fp}))$.
$O \twoheadleftarrow \mathcal{TSA}$: (σ_{fp})
O: $f_{\text{FP}}(fp) \coloneqq (com_{fp}, decom_{fp}, \sigma_{fp})$.
Output $\mathcal{TSA}$: Message $\texttt{ok}$
Output O: Message $\texttt{ok}$

UReg:
Input U: Message $\texttt{ureg}$, aux. input $input_U$
Input O: Message $\texttt{ureg}$, function parameters fp, aux. input $input_O$
U: Fetch verification keys:
– Call[a] $\mathcal{F}_{\text{BB}}$ instance $sid_{\mathcal{F}_{\text{BB}}}$ with input $(\texttt{Retrieve}, pid_O)$ and store output vk_O.
– Call[a] $\mathcal{F}_{\text{BB}}$ instance $sid_{\mathcal{F}_{\text{BB}}}$ with input $(\texttt{Retrieve}, pid_{\mathcal{TSA}})$ and store output $\mathsf{vk}_{\mathcal{TSA}}$.
Draw User secret key:
– Draw random User ID $id \xleftarrow{r} \mathbb{Z}_p$.
– Compute User public key $\mathsf{pk}_U \coloneqq id \cdot G_1$.
– Call $\mathcal{F}_{\text{BB}}$ instance $sid_{\mathcal{F}_{\text{BB}}}$ with input $(\texttt{Register}, pid_U, \mathsf{pk}_U)$.
– $(com_{id}, decom_{id}) \leftarrow \mathsf{COM}.\mathsf{commit}(id)$
Calculate proof of user secret key knowledge:
– $stmt \coloneqq (\mathsf{pk}_U, com_{id})$
– $wit \coloneqq (decom_{id}, id' \coloneqq id \cdot G_2)$
– $\Pi \leftarrow \mathsf{Prove}(stmt, wit, \mathcal{R}_{\mathsf{UR}})$, using $\mathcal{R}_{\mathsf{UR}}$ from Fig. 3.10.
Draw share of new serial number:
– $(ser^{\text{new}})^{(U)} \xleftarrow{r} \mathbb{Z}_p$
– $(com^{(U)}_{ser^{\text{new}}}, decom^{(U)}_{ser^{\text{new}}}) \leftarrow \mathsf{COM}.\mathsf{commit}((ser^{\text{new}})^{(U)})$
$U \twoheadrightarrow O$: $(\Pi, com_{id}, com^{(U)}_{ser^{\text{new}}})$
O: Fetch fp information:
– Set[a] $(com_{fp}, decom_{fp}, \sigma_{fp}) \coloneqq f_{\text{FP}}(fp)$.
O: Fetch user public key:
– Call $\mathcal{F}_{\text{BB}}$ instance $sid_{\mathcal{F}_{\text{BB}}}$ with input $(\texttt{Retrieve}, pid_U)$ and receive output pk_U.
Check[a] that no user with public key pk_U has registered yet.
Check proof:
– $stmt \coloneqq (\mathsf{pk}_U, com_{id})$
– Check[a] $\mathsf{Verify}(\Pi, stmt, \mathcal{R}_{\mathsf{UR}}) = 1$
$U \twoheadleftarrow O$: (com_{fp}, σ_{fp})
U: Check[a] that $\mathsf{SIG}'.\mathsf{Vf}((\texttt{ureg}, com_{fp}), \sigma_{fp}, \mathsf{vk}_{\mathcal{TSA}}) = 1$.
$U \leftrightarrow O$: Compute initial user history:
$U \rightarrow \mathcal{F}_{\text{PPA}}$: $(\texttt{ureg}, com_{fp}, input_U)$
$O \rightarrow \mathcal{F}_{\text{PPA}}$: $(\texttt{ureg}, fp, decom_{fp}, input_O)$
$U \leftarrow \mathcal{F}_{\text{PPA}}$: $(\mathcal{UH}, decom_{\mathcal{UH}}, output_U)$
$O \leftarrow \mathcal{F}_{\text{PPA}}$: $(com_{\mathcal{UH}}, output_O)$
O: Draw share of new serial number:
– $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
– $(com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}) \leftarrow \mathsf{COM}.\mathsf{commit}((ser^{\text{new}})^{(O)})$
Compute commitments and signature for initial User History:
– $com_{ser^{\text{new}}} \coloneqq com^{(O)}_{ser^{\text{new}}} \oplus com^{(U)}_{ser^{\text{new}}}$
– $(com_{lin}, decom_{lin}) \leftarrow \mathsf{COM}.\mathsf{commit}(0)$
– $\sigma \leftarrow \mathsf{SIG}.\mathsf{Sign}(\mathsf{sk}_O, com_{\mathcal{UH}} \| com_{ser^{\text{new}}} \| com_{lin} \| com_{id})$
$U \twoheadleftarrow O$: $(com_{\mathcal{UH}}, (ser^{\text{new}})^{(O)}, com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}, com_{lin}, decom_{lin}, \sigma)$
U: Set[a] and store $\mathsf{Log}^{\text{new}} \coloneqq \Pi_{\mathsf{Verify}}(\mathsf{VfyL}, (\mathcal{UH}, com_{\mathcal{UH}}, decom_{\mathcal{UH}}, ((ser^{\text{new}})^{(U)} + (ser^{\text{new}})^{(O)}),$
$(com^{(U)}_{ser^{\text{new}}} \oplus com^{(O)}_{ser^{\text{new}}}), (decom^{(U)}_{ser^{\text{new}}} \oplus decom^{(O)}_{ser^{\text{new}}}), 0, com_{lin}, decom_{lin}, id, com_{id}, decom_{id}, \sigma))$
Output U: Initial user history $\mathcal{UH}$, aux. output $output_U$
Output O: Aux. output $output_O$

$\{\text{Bookkeeping}^{(k)}\}_{k=1}^{K}$:
Input U: Message $\texttt{bookkeep}(k)$, aux. input $input_U$
Input O: Message $\texttt{bookkeep}(k)$, FPs fp, aux. input $input_O$
U: Calculate logbook-validity proof and draw share of new serial:
– $(\overline{com_{\mathcal{UH}}}, \overline{decom_{\mathcal{UH}}}) \leftarrow \text{COM.ReRand}(com_{\mathcal{UH}}, decom_{\mathcal{UH}})$
– $(\overline{com_{lin}}, \overline{decom_{lin}}) \leftarrow \text{COM.ReRand}(com_{lin}, decom_{lin})$
– $(\overline{com_{id}}, \overline{decom_{id}}) \leftarrow \text{COM.ReRand}(com_{id}, decom_{id})$
– $stmt_{Val} \coloneqq (\overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \mathsf{vk}_O)$
– $wit_{Val} \coloneqq$
$(com_{\mathcal{UH}}, decom_{\mathcal{UH}}, \overline{decom_{\mathcal{UH}}}, com_{ser}, decom_{ser}, com_{lin}, decom_{lin}, \overline{decom_{lin}}, \mathsf{pk}_U, com_{id}, decom_{id}, \overline{decom_{id}}, \sigma)$
– $\Pi_{Val} \leftarrow \text{POK.Prove}(stmt_{Val}, wit_{Val}, \mathcal{R}_{Val})$, using $\mathcal{R}_{Val}$ from Fig. 3.12a.
Draw share of new serial number:
– $(ser^{\text{new}})^{(U)} \xleftarrow{r} \mathbb{Z}_p$
– $(com_{ser^{\text{new}}}^{(U)}, decom_{ser^{\text{new}}}^{(U)}) \leftarrow \text{COM.commit}((ser^{\text{new}})^{(U)})$
$U \rightarrow O$: $(\overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \Pi_{Val}, com_{ser^{\text{new}}}^{(U)})$
O: Check proof and serial number and fetch fp information:
– $stmt_{Val} \coloneqq (\overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \mathsf{vk}_O)$
– Check[a] $\text{POK.Verify}(\Pi_{Val}, stmt_{Val}, \mathcal{R}_{Val}) = 1$
– Check[a] $ser \notin \mathsf{L}_{\mathsf{SER}}$
– $\mathsf{L}_{\mathsf{SER}} \coloneqq \mathsf{L}_{\mathsf{SER}} \cup \{ser\}$
– Set[a] $(com_{fp}, decom_{fp}, \sigma_{fp}) \coloneqq f_{\text{FP}}(fp)$.
$U \leftarrow O$: (com_{fp}, σ_{fp})
U: Check[a] that $\text{SIG}'.\text{Vf}((\texttt{bookkeep}(k), com_{fp}), \sigma_{fp}, \mathsf{vk}_{\mathcal{TSA}}) = 1$.
Communicate with $\mathcal{F}_{\text{PPA}}$:
$U \rightarrow \mathcal{F}_{\text{PPA}}$: $(\texttt{bookkeep}(k), \mathcal{UH}, \overline{decom_{\mathcal{UH}}}, com_{fp}, input_U)$
$O \rightarrow \mathcal{F}_{\text{PPA}}$: $(\texttt{bookkeep}(k), \overline{com_{\mathcal{UH}}}, fp, decom_{fp}, input_O)$
$U \leftarrow \mathcal{F}_{\text{PPA}}$: $(\alpha, \mathbf{s}, \mathbf{a}, com_{\mathbf{a}}, decom_{\mathbf{a}}, output_U)$
$O \leftarrow \mathcal{F}_{\text{PPA}}$: $(\alpha, \mathbf{s}, com_{\mathbf{a}}, output_O)$
U: Calculate new User History and prove correctness:
– **if** $\alpha \neq \perp \vee \mathbf{s} \neq \perp$ **then**
* Apply permutation: $\mathcal{UH}' \leftarrow \alpha(\mathcal{UH})$.
* For i from 0 to $|\mathcal{UH}| - 1$, if $\mathbf{s}[i] \neq \perp$, then set $\mathcal{UH}''[i] \coloneqq \mathbf{s}[i]$, else copy $\mathcal{UH}''[i] \coloneqq \mathcal{UH}'[i]$.
* Apply addition: $\mathcal{UH}^{\text{new}} \leftarrow \mathcal{UH}'' + \mathbf{a}$.
* $(com'_{\mathcal{UH}}, decom'_{\mathcal{UH}}) \leftarrow \text{COM.commit}(\mathcal{UH}')$.
* $(com''_{\mathcal{UH}}, decom''_{\mathcal{UH}}) \leftarrow \text{COM.commit}(\mathcal{UH}'')$.
* $(com_{\mathcal{UH}}^{\text{new}}, decom_{\mathcal{UH}}^{\text{new}}) \leftarrow (com''_{\mathcal{UH}}, decom''_{\mathcal{UH}}) \oplus (com_{\mathbf{a}}, decom_{\mathbf{a}})$.
* Parse $(uh_0, \ldots, uh_{m-1}) \coloneqq \mathcal{UH}$, $(uh'_0, \ldots, uh'_{m-1}) \coloneqq \mathcal{UH}'$, and $(uh''_0, \ldots, uh''_{m-1}) \coloneqq \mathcal{UH}''$.
* $stmt_{Tr} \coloneqq (\overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \alpha, \mathbf{s})$
* $wit_{Tr} \coloneqq (\overline{decom_{\mathcal{UH}}}, decom'_{\mathcal{UH}}, decom''_{\mathcal{UH}}, uh_0, \ldots, uh_{m-1}, uh'_0, \ldots, uh'_{m-1}, uh''_0, \ldots, uh''_{m-1})$
* $\Pi_{Tr} \leftarrow \text{POK.Prove}(stmt_{Tr}, wit_{Tr}, \mathcal{R}_{Tr})$, using $\mathcal{R}_{Tr}$ from Fig. 3.12b.
$\boldsymbol{U \rightarrow O}$: $\boldsymbol{(com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \Pi_{Tr})}$
– **else**
* $(com_{\mathcal{UH}}^{\text{new}}, decom_{\mathcal{UH}}^{\text{new}}) \leftarrow (com''_{\mathcal{UH}}, decom''_{\mathcal{UH}}) \oplus (com_{\mathbf{a}}, decom_{\mathbf{a}})$.
O: Draw share of new serial number:
– $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
– $(com_{ser^{\text{new}}}^{(O)}, decom_{ser^{\text{new}}}^{(O)}) \leftarrow \text{COM.commit}((ser^{\text{new}})^{(O)})$
Verify proof, compute signature for the new UH.
– **if** $\alpha \neq \perp \vee \mathbf{s} \neq \perp$ **then**
* $stmt_{Tr} \coloneqq (\overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \alpha, \mathbf{s})$
* Check[a] $\text{POK.Verify}(\Pi_{Tr}, stmt_{Tr}, \mathcal{R}_{Tr}) = 1$.
* $com_{\mathcal{UH}}^{\text{new}} \coloneqq com''_{\mathcal{UH}} \oplus com_{\mathbf{a}}$.
– **else**
* $com_{\mathcal{UH}}^{\text{new}} \coloneqq \overline{com_{\mathcal{UH}}} \oplus com_{\mathbf{a}}$.
– $com_{ser}^{\text{new}} \coloneqq com_{ser^{\text{new}}}^{(O)} \oplus com_{ser^{\text{new}}}^{(U)}$
– $\sigma^{\text{new}} \leftarrow \text{SIG.Sign}(\mathsf{sk}_O, com_{\mathcal{UH}}^{\text{new}} \| com_{ser}^{\text{new}} \| \overline{com_{lin}} \| \overline{com_{id}})$
$U \leftarrow O$: $(com_{\mathcal{UH}}^{\text{new}}, (ser^{\text{new}})^{(O)}, com_{ser^{\text{new}}}^{(O)}, decom_{ser^{\text{new}}}^{(O)}, \sigma^{\text{new}})$
U: Set[a] and store $\text{Log}^{\text{new}} \coloneqq \Pi_{\text{Verify}}(\text{VfyL}, (\mathcal{UH}^{\text{new}}, com_{\mathcal{UH}}^{\text{new}}, decom_{\mathcal{UH}}^{\text{new}}, ((ser^{\text{new}})^{(U)} \oplus (ser^{\text{new}})^{(O)}),$
$(com_{ser^{\text{new}}}^{(U)} \oplus com_{ser^{\text{new}}}^{(O)}), (decom_{ser^{\text{new}}}^{(U)} \oplus decom_{ser^{\text{new}}}^{(O)}), lin, \overline{com_{lin}}, \overline{decom_{lin}}, id, \overline{com_{id}}, \overline{decom_{id}}, \sigma^{\text{new}}))$.
Output U: User History $\mathcal{UH}^{\text{new}}$, aux. output $output_U$.
Output O: Aux. output $output_O$, permutation α, set vector $\mathbf{s}$.

$\{\text{Outsource}^{(k)}\}_{k=1}^{K}$:
Input U: Message $\texttt{outsource}k$, aux. input for the computation $input_U$.
Input P and O: Message $\texttt{outsource}k$

$\boldsymbol{U}$: Create robust secret shares:
- $(\mathsf{o}_{\text{OUT}}, \mathsf{o}_{\text{UNV}}, \mathsf{o}_\alpha, \mathsf{o}_\mathsf{s}, \mathsf{o}_\mathsf{a}, \{\langle\mathcal{UH}\rangle^x, \langle input_U\rangle^x, \langle\mathsf{o}_{\text{OUT}}\rangle^x, \langle\mathsf{o}_{\text{UNV}}\rangle^x, \langle\mathsf{o}_\alpha\rangle^x, \langle\mathsf{o}_\mathsf{s}\rangle^x, \langle\mathsf{o}_\mathsf{a}\rangle^x\}_{x\in\{P,O\}}) \leftarrow \Pi_{\text{Verify}}(\texttt{ShareL}, \text{Log})$

Calculate proof and draw serial number share:
- $(\overline{com_{id}}, \overline{decom_{id}}) \leftarrow \text{COM.ReRand}(com_{id}, decom_{id})$
- $stmt \coloneqq (\langle\mathcal{UH}\rangle^O, com^{(O)}_{\mathcal{UH}}, ser, \overline{com_{id}}, \mathsf{vk}_O)$
- $wit \coloneqq (com_{\mathcal{UH}}, decom_{\mathcal{UH}}, decom^{(P)}_{\mathcal{UH}}, com_{ser}, decom_{ser}, com_{lin}, decom_{lin}, \mathsf{pk}_U, com_{id}, decom_{id}, \overline{decom_{id}}, \sigma)$
- $\Pi \leftarrow \text{POK.Prove}(stmt, wit, \mathcal{R}_{\text{0S}})$, using $\mathcal{R}_{\text{0S}}$ from Fig. 3.14a.
- $(ser^{\text{new}})^{(U)} \xleftarrow{\text{r}} \mathbb{Z}_p$.
- $(com^{(U)}_{ser^{\text{new}}}, decom^{(U)}_{ser^{\text{new}}}) \leftarrow \text{COM.commit}((ser^{\text{new}})^{(U)})$

$\boldsymbol{U}\rightarrow\boldsymbol{O}$: $(\langle\mathcal{UH}\rangle^O, \langle input_U\rangle^O, \langle\mathsf{o}_{\text{OUT}}\rangle^O, \langle\mathsf{o}_{\text{UNV}}\rangle^O, \langle\mathsf{o}_\alpha\rangle^O, \langle\mathsf{o}_\mathsf{s}\rangle^O, \langle\mathsf{o}_\mathsf{a}\rangle^O, ser, \overline{com_{id}}, \Pi, com^{(U)}_{ser^{\text{new}}})$

$\boldsymbol{U}\rightarrow\boldsymbol{P}$: $(\langle\mathcal{UH}\rangle^P, \langle input_U\rangle^P, \langle\mathsf{o}_{\text{OUT}}\rangle^P, \langle\mathsf{o}_{\text{UNV}}\rangle^P, \langle\mathsf{o}_\alpha\rangle^P, \langle\mathsf{o}_\mathsf{s}\rangle^P, \langle\mathsf{o}_\mathsf{a}\rangle^P)$

$\boldsymbol{O}$: Check[a] serial number $ser \notin \mathsf{L}_{\text{SER}}$ and add ser to L_{SER}.

Prepare linking number: $(lin^{\text{new}})^{(O)} \xleftarrow{\text{r}} \mathbb{Z}_p$

$(com^{(O)}_{lin^{\text{new}}}, decom^{(O)}_{lin^{\text{new}}}) \leftarrow \text{commit}((lin^{\text{new}})^{(O)})$

$\boldsymbol{O}\twoheadrightarrow\boldsymbol{P}$: $(com^{(O)}_{lin^{\text{new}}}, com^{(P)}_{\mathcal{UH}}, com^{(P)}_{input_U}, com^{(P)}_{\mathsf{o}_{\text{OUT}}}, com^{(P)}_{\mathsf{o}_{\text{UNV}}}, com^{(P)}_{\mathsf{o}_\alpha}, com^{(P)}_{\mathsf{o}_\mathsf{s}}, com^{(P)}_{\mathsf{o}_\mathsf{a}})$

$\boldsymbol{P}$: Check[a] $\Pi_{\text{Verify}}(\texttt{VfyS}, com^{(P)}_{\mathcal{UH}}, decom^{(P)}_{\mathcal{UH}}, \mathcal{UH}^{(P)}, com^{(P)}_{input_U}, decom^{P}_{input_U}, input^{P}_{U}, com^{(P)}_{\mathsf{o}_{\text{OUT}}}, decom^{P}_{\mathsf{o}_{\text{OUT}}}, \mathsf{o}_{\text{OUT}}{}^{P}, com^{(P)}_{\mathsf{o}_{\text{UNV}}}, decom^{P}_{\mathsf{o}_{\text{UNV}}}, \mathsf{o}_{\text{UNV}}{}^{P}, com^{(P)}_{\mathsf{o}_\alpha}, decom^{(P)}_{\mathsf{o}_\alpha}, \mathsf{o}_\alpha{}^{(P)}, com^{(P)}_{\mathsf{o}_\mathsf{s}}, decom^{(P)}_{\mathsf{o}_\mathsf{s}}, \mathsf{o}_\mathsf{s}{}^{(P)}, com^{(P)}_{\mathsf{o}_\mathsf{a}}, decom^{(P)}_{\mathsf{o}_\mathsf{a}}, \mathsf{o}_\mathsf{a}{}^{(P)}) = 1$

Draw $(lin^{\text{new}})^{(P)} \xleftarrow{\text{r}} \mathbb{Z}_p$

$\boldsymbol{P}\twoheadrightarrow\boldsymbol{O}$: $((lin^{\text{new}})^{(P)}, com^{(O)}_{\mathcal{UH}}, com^{(O)}_{input_U}, com^{(O)}_{\mathsf{o}_{\text{OUT}}}, com^{(O)}_{\mathsf{o}_{\text{UNV}}}, com^{(O)}_{\mathsf{o}_\alpha}, com^{(O)}_{\mathsf{o}_\mathsf{s}}, com^{(O)}_{\mathsf{o}_\mathsf{a}})$

$\boldsymbol{O}$: Check proof and sharings
- $stmt \coloneqq (\langle\mathcal{UH}\rangle^O, com^{(O)}_{\mathcal{UH}}, ser, \overline{com_{id}}, \mathsf{vk}_O)$
- Check[a] $\text{POK.Verify}(\Pi, stmt, \mathcal{R}_{\text{0S}}) = 1$
- Check[a] $\Pi_{\text{Verify}}(\texttt{VfyS}, com^{(O)}_{\mathcal{UH}}, decom^{(O)}_{\mathcal{UH}}, \mathcal{UH}^{(O)}, com^{(O)}_{input_U}, decom^{O}_{input_U}, input^{O}_{U}, com^{(O)}_{\mathsf{o}_{\text{OUT}}}, decom^{O}_{\mathsf{o}_{\text{OUT}}}, \mathsf{o}_{\text{OUT}}{}^{O}, com^{(O)}_{\mathsf{o}_{\text{UNV}}}, decom^{O}_{\mathsf{o}_{\text{UNV}}}, \mathsf{o}_{\text{UNV}}{}^{O}, com^{(O)}_{\mathsf{o}_\alpha}, decom^{(O)}_{\mathsf{o}_\alpha}, \mathsf{o}_\alpha{}^{(O)}, com^{(O)}_{\mathsf{o}_\mathsf{s}}, decom^{(O)}_{\mathsf{o}_\mathsf{s}}, \mathsf{o}_\mathsf{s}{}^{(O)}, com^{(O)}_{\mathsf{o}_\mathsf{a}}, decom^{(O)}_{\mathsf{o}_\mathsf{a}}, \mathsf{o}_\mathsf{a}{}^{(O)}) = 1$

Store outsource information and draw share of new serial number:
- $lin^{\text{new}} \coloneqq (lin^{\text{new}})^{(O)} + (lin^{\text{new}})^{(P)}$
- Add $(lin^{\text{new}}, \langle\mathcal{UH}\rangle^O, \langle input_U\rangle^O, \langle\mathsf{o}_{\text{OUT}}\rangle^O, \langle\mathsf{o}_{\text{UNV}}\rangle^O, \langle\mathsf{o}_\alpha\rangle^O, \langle\mathsf{o}_\mathsf{s}\rangle^O, \langle\mathsf{o}_\mathsf{a}\rangle^O)$ to $f^{(O)}_{\text{OI}}(pid_P, k)$
- $(ser^{\text{new}})^{(O)} \xleftarrow{\text{r}} \mathbb{Z}_p$
- $(com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}) \leftarrow \text{COM.commit}((ser^{\text{new}})^{(O)})$

Compute commitments and signature for updated UH:
- $com^{\text{new}}_{\mathcal{UH}} \coloneqq com^{(P)}_{\mathcal{UH}} \oplus com^{(O)}_{\mathcal{UH}}$
- $com^{\text{new}}_{ser} \coloneqq com^{(U)}_{ser^{\text{new}}} \oplus com^{(O)}_{ser^{\text{new}}}$
- $(com^{\text{new}}_{lin}, decom^{\text{new}}_{lin}) \leftarrow \text{COM.commit}(lin^{\text{new}})$
- $\sigma^{\text{new}} \leftarrow \text{SIG.Sign}(\mathsf{sk}_O, com^{\text{new}}_{\mathcal{UH}} \| com^{\text{new}}_{ser} \| com^{\text{new}}_{lin} \| \overline{com_{id}})$

$\boldsymbol{U}\leftarrow\boldsymbol{O}$: $((ser^{\text{new}})^{(O)}, com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}, com^{\text{new}}_{lin}, decom^{\text{new}}_{lin}, \sigma^{\text{new}})$

$\boldsymbol{P}\twoheadleftarrow\boldsymbol{O}$: $((lin^{\text{new}})^{(O)}, decom^{(O)}_{lin^{\text{new}}})$

$\boldsymbol{P}$: Store outsource information:
- Check[a] $\text{open}(com^{(O)}_{lin^{\text{new}}}, decom^{(O)}_{lin^{\text{new}}}, (lin^{\text{new}})^{(O)}) = 1$
- $lin^{\text{new}} \coloneqq (lin^{\text{new}})^{(O)} + (lin^{\text{new}})^{(P)}$
- Add $(lin^{\text{new}}, \langle\mathcal{UH}\rangle^P, \langle input_U\rangle^P, \langle\mathsf{o}_{\text{OUT}}\rangle^P, \langle\mathsf{o}_{\text{UNV}}\rangle^P, \langle\mathsf{o}_\alpha\rangle^P, \langle\mathsf{o}_\mathsf{s}\rangle^P, \langle\mathsf{o}_\mathsf{a}\rangle^P)$ to $f^{(P)}_{\text{OI}}(k)$

$\boldsymbol{U}\leftarrow\boldsymbol{P}$: (lin^{new})

$\boldsymbol{U}$: Set[a] and store $\text{Log}^{\text{new}} \coloneqq \Pi_{\text{Verify}}(\texttt{VfyL}, (\mathcal{UH}, (com^{(P)}_{\mathcal{UH}} \oplus com^{(O)}_{\mathcal{UH}}), (decom^{(P)}_{\mathcal{UH}} \oplus decom^{(O)}_{\mathcal{UH}}), ((ser^{\text{new}})^{(U)} + (ser^{\text{new}})^{(O)}), (com^{(U)}_{ser^{\text{new}}} \oplus com^{(O)}_{ser^{\text{new}}}), (decom^{(U)}_{ser^{\text{new}}} \oplus decom^{(O)}_{ser^{\text{new}}}), lin^{\text{new}}, com^{\text{new}}_{lin}, decom^{\text{new}}_{lin}, id, \overline{com_{id}}, \overline{decom_{id}}, \sigma^{\text{new}}))$.

Output $\boldsymbol{U}$, $\boldsymbol{P}$ and $\boldsymbol{O}$: Confirmation ok.

$\{\text{Analytics}^{(k)}\}_{k=1}^{K}$:

Input $\boldsymbol{P}$: Message $\texttt{analyt}k$.

Input $\boldsymbol{O}$: Message $\texttt{analyt}k$, FPs fp, aux. input $input_O$.

$\boldsymbol{P}$: Load outsource information:
- Load[a] and remove the first Z_k entries $\{(lin, \langle\mathcal{UH}\rangle^P, \langle input_U\rangle^P, \langle\mathsf{o}_{\text{OUT}}\rangle^P, \langle\mathsf{o}_{\text{UNV}}\rangle^P, \langle\mathsf{o}_\alpha\rangle^P, \langle\mathsf{o}_\mathsf{s}\rangle^P, \langle\mathsf{o}_\mathsf{a}\rangle^P)_z\}_{z=1}^{Z_k}$ from $f^{(P)}_{\text{OI}}(k)$.

$\boldsymbol{O}$: Load outsource information:
- Load[a] and remove the first Z_k entries $\{(lin, \langle\mathcal{UH}\rangle^O, \langle input_U\rangle^O, \langle\mathsf{o}_{\text{OUT}}\rangle^O, \langle\mathsf{o}_{\text{UNV}}\rangle^O, \langle\mathsf{o}_\alpha\rangle^O, \langle\mathsf{o}_\mathsf{s}\rangle^O, \langle\mathsf{o}_\mathsf{a}\rangle^O)_z\}_{z=1}^{Z_k}$ from $f^{(O)}_{\text{OI}}(pid_P, k)$.

O: Fetch fp information:
– Set[a] $(com_{fp},\ decom_{fp},\ \sigma_{fp}) \coloneqq f_{\text{FP}}(fp)$.
O*→*P: $(com_{fp},\ \sigma_{fp})$
P: Check[a] that $\text{SIG}'.\text{Vf}((\texttt{analyt}k, com_{fp}),\ \sigma_{fp},\ \text{vk}_{\mathcal{TSA}}) = 1$.
P*⇹*O: Compute function:
***P*→$\mathcal{F}_{\text{PPA}}$**: $(\texttt{analyt}k,\ com_{fp},\ \{(\langle\mathcal{UH}\rangle^P, \langle input_U\rangle^P, \langle \text{o}_{\text{OUT}}\rangle^P, \langle \text{o}_{\text{UNV}}\rangle^P, \langle \text{o}_\alpha\rangle^P, \langle \text{o}_\text{s}\rangle^P, \langle \text{o}_\text{a}\rangle^P)_z\}_{z=1}^{Z_k})$
***O*→$\mathcal{F}_{\text{PPA}}$**: $(\texttt{analyt}k,\ fp,\ decom_{fp},\ \{(\langle\mathcal{UH}\rangle^O, \langle input_U\rangle^O, \langle \text{o}_{\text{OUT}}\rangle^O, \langle \text{o}_{\text{UNV}}\rangle^O, \langle \text{o}_\alpha\rangle^O, \langle \text{o}_\text{s}\rangle^O, \langle \text{o}_\text{a}\rangle^O)_z\}_{z=1}^{Z_k},\ input_O)$
***P*←$\mathcal{F}_{\text{PPA}}$**: $\{(ct_{\alpha_z},\ ct_{\text{s}_z},\ ct_{\text{a}_z},\ ct_{decom_{\text{a}_z}},\ ct_{output_{U_z}})\}_{z=1}^{Z_k}$
***O*←$\mathcal{F}_{\text{PPA}}$**: $\{(\alpha,\ \text{s},\ com_{\text{a}_z},\ ct_{output_{U_z}})\}_{z=1}^{Z_k}, output_O$
O: Store update information:
For all $z \in (1,\ \ldots,\ Z_k)$: set $f_{\text{UI}}^{(O)}(lin_z) \coloneqq (\alpha,\ \text{s},\ com_\text{a},\ ct_{output_U})_z$.
P: Store update information:
For all $z \in (1, \ldots, Z_k)$: set $f_{\text{UI}}^{(P)}(lin_z) \coloneqq (ct_\alpha,\ ct_\text{s},\ ct_\text{a},\ ct_{decom_\text{a}},\ ct_{output_U})_z$
Output *P*: Confirmation ok.
Output *O*: Aux. output $output_O$.

Update:
Input *U*: update
Input *P*: update
Input *O*: update
U*→*P: (lin)
P: Set[a] and remove $(ct_\alpha,\ ct_\text{s},\ ct_\text{a},\ ct_{decom_\text{a}},\ ct_{output_U}) \coloneqq f_{\text{UI}}^{(P)}(lin)$
U*←*P: $(ct_\alpha,\ ct_\text{s},\ ct_\text{a},\ ct_{decom_\text{a}},\ ct_{output_U})$
U: Prove logbook validity and correct updates and draw share of serial number:
– $(\overline{com_{\mathcal{UH}}},\ \overline{decom_{\mathcal{UH}}}) \leftarrow \text{COM.ReRand}(com_{\mathcal{UH}},\ decom_{\mathcal{UH}})$
– Set $\alpha \coloneqq ct_\alpha - \text{o}_\alpha$, $\text{s} \coloneqq ct_\text{s} - \text{o}_\text{s}$ and $\text{a} \coloneqq ct_\text{a} - \text{o}_\text{a}$.
– Set $decom_\text{a} \coloneqq ct_{decom_\text{a}} - \text{o}_{\text{UNV}}$ and $output_U \coloneqq ct_{output_U} - \text{o}_{\text{OUT}}$
– $(\overline{com_{id}},\ \overline{decom_{id}}) \leftarrow \text{COM.ReRand}(com_{id},\ decom_{id})$
– $stmt \coloneqq (\overline{com_{\mathcal{UH}}},\ ser,\ lin,\ \overline{com_{id}},\ \text{vk}_O)$
– $wit \coloneqq (com_{\mathcal{UH}},\ decom_{\mathcal{UH}},\ \overline{decom_{\mathcal{UH}}},\ com_{ser},\ decom_{ser},\ com_{lin},\ decom_{lin},\ \text{pk}_U,\ com_{id},\ decom_{id},\ \overline{decom_{id}},\ \sigma)$
– $\Pi \leftarrow \text{POK.Prove}(stmt,\ wit,\ \mathcal{R}_{Upd})$, using $\mathcal{R}_{Upd}$ from Fig. 3.14b.
– $(ser^{\text{new}})^{(U)} \xleftarrow{r} \mathbb{Z}_p$
– $(com_{ser^{\text{new}}}^{(U)},\ decom_{ser^{\text{new}}}^{(U)}) \leftarrow \text{COM.commit}((ser^{\text{new}})^{(U)})$
– **if** $\alpha \neq \perp \wedge \text{s} \neq \perp$ **then**
* Apply permutation: $\mathcal{UH}' \leftarrow \alpha(\overline{\mathcal{UH}})$.
* For i from 0 to $|\mathcal{UH}| - 1$, if $\text{s}[i] \neq \perp$, then set value $\mathcal{UH}''[i] \coloneqq \text{s}[i]$, else copy $\mathcal{UH}''[i] \coloneqq \mathcal{UH}'[i]$.
* Apply addition: $\mathcal{UH}^{\text{new}} \leftarrow \mathcal{UH}'' + \text{a}$.
* $(com'_{\mathcal{UH}},\ decom'_{\mathcal{UH}}) \leftarrow \text{COM.commit}(\mathcal{UH}')$.
* $(com''_{\mathcal{UH}},\ decom''_{\mathcal{UH}}) \leftarrow \text{COM.commit}(\mathcal{UH}'')$.
* $stmt_{Tr} \coloneqq (\overline{com_{\mathcal{UH}}},\ com'_{\mathcal{UH}},\ com''_{\mathcal{UH}},\ \alpha,\ \text{s})$
* $wit_{Tr} \coloneqq (\overline{decom_{\mathcal{UH}}},\ decom'_{\mathcal{UH}},\ decom''_{\mathcal{UH}},\ uh_0,\ \ldots,\ uh_{m-1},\ uh'_0,\ \ldots,\ uh'_{m-1},\ uh''_0,\ \ldots,\ uh''_{m-1})$
* $\Pi_{Tr} \leftarrow \text{POK.Prove}(stmt_{Tr},\ wit_{Tr},\ \mathcal{R}_{Tr})$, using $\mathcal{R}_{Tr}$ from Fig. 3.12b.
U*→*O: $(\boldsymbol{\overline{com_{\mathcal{UH}}}},\ \boldsymbol{com'_{\mathcal{UH}}},\ \boldsymbol{com''_{\mathcal{UH}}},\ \boldsymbol{\overline{com_{id}}},\ \boldsymbol{ser},\ \boldsymbol{lin},\ \Pi,\ \boldsymbol{\Pi_{Tr}},\ \boldsymbol{com_{ser^{\text{new}}}^{(U)}})$
– **else**
* Apply addition: $\mathcal{UH}^{\text{new}} \leftarrow \mathcal{UH} + \text{a}$.
U*→*O: $(\boldsymbol{\overline{com_{\mathcal{UH}}}},\ \boldsymbol{\overline{com_{id}}},\ \boldsymbol{ser},\ \boldsymbol{lin},\ \Pi,\ \boldsymbol{com_{ser^{\text{new}}}^{(U)}})$
O: Load update information:
– Set[a] $(\alpha,\ \text{s},\ com_\text{a},\ (ct_{output_U})') \coloneqq f_{\text{UI}}^{(O)}(lin)$
– Remove $lin \mapsto (\alpha,\ \text{s},\ com_\text{a},\ (ct_{output_U})')$ from $f_{\text{UI}}^{(O)}$
Check proof and serial number:
– $stmt \coloneqq (\overline{com_{\mathcal{UH}}},\ ser,\ lin,\ \overline{com_{id}},\ \text{vk}_O)$
– Check[a] $\text{POK.Verify}(\Pi,\ stmt,\ \mathcal{R}_{Upd}) \stackrel{?}{=} 1$
– **if** $\alpha \neq \perp \vee \text{s} \neq \perp$ **then**
* $stmt_{Tr} \coloneqq (\overline{com_{\mathcal{UH}}},\ com'_{\mathcal{UH}},\ com''_{\mathcal{UH}},\ \alpha,\ \text{s})$
* Check[a] $\text{POK.Verify}(\Pi_{Tr},\ stmt_{Tr},\ \mathcal{R}_{Tr}) \stackrel{?}{=} 1$
* $com_{\mathcal{UH}}^{\text{new}} \coloneqq com''_{\mathcal{UH}} \oplus com_\text{a}$
– **else**
* $com_{\mathcal{UH}}^{\text{new}} \coloneqq \overline{com_{\mathcal{UH}}} \oplus com_\text{a}$
– Check[a] $ser \notin \text{L}_{\text{SER}}$
– $\text{L}_{\text{SER}} \coloneqq \text{L}_{\text{SER}} \cup \{ser\}$
Draw share of new serial number:
– $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$

– $(com_{ser^{\text{new}}}^{(O)},\ decom_{ser^{\text{new}}}^{(O)}) \leftarrow \mathsf{COM.commit}((ser^{\text{new}})^{(O)})$

Compute commitments and signature for updated UH:

– $com_{ser}^{\text{new}} \coloneqq com_{ser^{\text{new}}}^{(O)} \oplus com_{ser^{\text{new}}}^{(U)}$

– $(com_{lin}^{\text{new}},\ decom_{lin}^{\text{new}}) \leftarrow \mathsf{COM.commit}(0)$

– $\sigma^{\text{new}} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O,\ com_{\mathcal{UH}}^{\text{new}} \| com_{ser}^{\text{new}} \| com_{lin}^{\text{new}} \| \overline{com_{id}})$

$U \leftarrow O$: $(com_{\mathbf{a}},\ com_{\mathcal{UH}}^{\text{new}},\ (ser^{\text{new}})^{(O)},\ com_{ser^{\text{new}}}^{(O)},\ decom_{ser^{\text{new}}}^{(O)},\ com_{lin}^{\text{new}},\ decom_{lin}^{\text{new}},\ \sigma^{\text{new}},\ ct_{output_U})$

U: Check[a] $\mathsf{open}(com_{\mathbf{a}},\ decom_{\mathbf{a}},\ \mathbf{a}) \stackrel{?}{=} 1$ and[a] $ct_{output_U}{}' \stackrel{?}{=} ct_{output_U}$ Set[a] and store $\mathsf{Log}^{\text{new}} \coloneqq \Pi_{\text{Verify}}(\mathtt{VfyL}, (\mathcal{UH}^{\text{new}},\ com_{\mathcal{UH}}^{\text{new}},\ (\overline{decom_{\mathcal{UH}}} \oplus decom_{\mathbf{a}}),\ ((ser^{\text{new}})^{(U)} + (ser^{\text{new}})^{(O)}),\ (com_{ser^{\text{new}}}^{(U)} \oplus com_{ser^{\text{new}}}^{(O)}),\ (decom_{ser^{\text{new}}}^{(U)} \oplus decom_{ser^{\text{new}}}^{(O)}),\ 0,\ com_{lin}^{\text{new}},\ decom_{lin}^{\text{new}},\ id,\ \overline{com_{id}},\ \overline{decom_{id}},\ \sigma^{\text{new}}))$

Output U: Permutation α, set vector $\mathbf{s}$, add vector $\mathbf{a}$, aux. $output_U$.

Output P: Confirmation ok.

Output O: Permutation α, set vector $\mathbf{s}$.

[a] If this fails, output $\perp$ and abort.

A.3.1. User Security

In this section, we investigate the security of our system in scenarios that relate to the security of an honest user U. To that end, we prove the following theorem:

Theorem A.3.1 (User Security). *If instantiated with a trapdoor-commitment scheme* COM *and a dual-mode zero-knowledge protocol* POK*, it holds that*

$$\Pi_{\text{PUBA}}^{(\mathcal{F}_{\text{PPA}},\ \mathcal{F}_{\text{BB}},\ \mathcal{F}_{\text{KE}},\ \mathcal{F}_{\text{CRS}})} \geq_{UC} \mathcal{F}_{\text{PUBA}}$$

against all PPT-adversaries $\mathcal{A}$ who statically corrupted the operator O and a subset of users U.

We use the UC-framework [Can01] and provide a simulator $\mathcal{S}$ for this case. The simulator provides a view for any PPT-environment $\mathcal{Z}$ (that is restricted to not corrupting any proxies) that is consistent with a *real* protocol execution.

The simulator is given as follows:

Simulator $\mathcal{S}_{\text{USec}}$ for a corrupted operator

Π-*Shared State*:

- Trapdoor td_{sim} for simulating proofs.
- Verification key vk_O of the operator
- Mapping f_{OI} on $\{pid_P\} \times \{1, \ldots, K\}$ that maps $(pid_P,\ k)$ to a list $f_{\text{OI}}(pid_P,\ k)$ of entries $(lin,\ \langle \mathcal{UH} \rangle^P,\ \langle \mathtt{in}_U \rangle^P,\ \langle \mathsf{o}_{\text{OUT}} \rangle^P,\ \langle \mathsf{o}_{\text{UNV}} \rangle^P,\ \langle \mathsf{o}_\alpha \rangle^P,\ \langle \mathsf{o}_{\mathbf{s}} \rangle^P,\ \langle \mathsf{o}_{\mathbf{a}} \rangle^P)$ for all proxies P
- Partial mapping f_{UI} on $\{lin\}$: $lin \mapsto (com_{\mathbf{a}_z},\ decom_{\mathbf{a}_z},\ \mathbf{a}_z,\ \alpha,\ \mathbf{s},\ \mathsf{o}_\alpha,\ \mathsf{o}_{\mathbf{s}},\ \mathsf{o}_{\mathbf{a}},\ \mathsf{o}_{\text{UNV}},\ ct_{\mathtt{out}_z})$
- Partial mapping f_{LN} on $\{ssid\}$: $ssid \mapsto lin$
- Mapping f_{FP} that maps a given task $\mathtt{task}$ to a set of tuples $\{fp, com_{fp}, \sigma_{fp}\}$ of function parameters and corresponding signatures.

Π_{PUBA}-*Setup*:

1. Run a modified version of $crs \leftarrow \Pi_{\text{PUBA}}\text{-}Setup(1^\lambda)$:
   - $crs_{\text{ZK}} \leftarrow$ SetupPoK is replaced by $(crs_{\text{ZK}},\ td_{\text{sim}}) \leftarrow$ SetupSPoK for simulating proofs.

Π_{PUBA}-*Init*:

1. Upon receiving instructions from $\mathcal{Z}$ to send $(\mathtt{Register},\ pid_O,\ \mathsf{vk}_O)$ from O to $\mathcal{F}_{\text{BB}}$:
   - Check[a] that no key has previously been stored for pid_O.

- Store vk_O.

↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{init})$ in the name of O.

2. Upon receiving output $(\texttt{ok})$ from $\mathcal{F}_{\text{PUBA}}$ to O:
   ↪ Report message $(\texttt{ok})$ from $\mathcal{F}_{\text{BB}}$ to O.
3. Upon receiving $(\texttt{init}, pid_{\mathcal{TSA}})$ from $\mathcal{F}_{\text{PUBA}}$:
   - Generate signature key pair: $(\mathsf{vk}_{\mathcal{TSA}}, \mathsf{sk}_{\mathcal{TSA}}) \leftarrow \mathsf{SIG.Keygen}(\mathsf{gp})$
   - Respond to future calls of the form $(\texttt{Retrieve}, pid_{\mathcal{TSA}})$ to $\mathcal{F}_{\text{BB}}$ with $\mathsf{sk}_{\mathcal{TSA}}$.

Π_{PUBA}-*SignFP*:

1. Upon receiving instructions from $\mathcal{Z}$ to send $(fp, com_{fp}, decom_{fp}, \texttt{task}, \texttt{in}_O)$ from O to $\mathcal{TSA}$:
   - Check[a] that $\mathsf{COM.open}(com_{fp}, decom_{fp}, fp) = 1$.

   ↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ in the name of O with input $(\texttt{signfp}, fp, \texttt{task}, \texttt{in}_O)$.
2. Upon receiving output ok from $\mathcal{F}_{\text{PUBA}}$ to O:
   - Compute signature $\mathsf{SIG.Sign}(\mathsf{sk}_{\mathcal{TSA}}, (\texttt{task}, com_{fp}))$.
   - Add $(fp, com_{fp}, \sigma_{fp})$ to $f_{\text{FP}}(\texttt{task})$.

   ↪ Report message (σ_{fp}) from $\mathcal{TSA}$ to O.

Π_{PUBA}-*UReg*:

. . . . . .

U honest, O corrupted:

1. Upon receiving $(\texttt{ureg}, pid_U)$ from $\mathcal{F}_{\text{PUBA}}$:
   - $id \xleftarrow{r} \mathbb{Z}_p, \mathsf{pk}_U := id \cdot G_1$
   - $(com_{id}, decom_{id}) \leftarrow \mathsf{COM.commit}(\mathbf{0})$
   - $stmt := (\mathsf{pk}_U, com_{id})$
   - $\Pi \leftarrow \mathsf{SimProof}(stmt, \mathcal{R}_{\mathsf{UR}})$
   - $(com^{(U)}_{ser^{\text{new}}}, decom^{(U)}_{ser^{\text{new}}}) \leftarrow \mathsf{COM.commit}(\mathbf{0})$

   ↪ Report message $(\Pi, com_{id}, com^{(U)}_{ser^{\text{new}}})$ from U to O.
2. Upon receiving $(\texttt{Retrieve}, pid_U)$ from O to $\mathcal{F}_{\text{BB}}$:
   ↪ Report message $(\texttt{Retrieve}, pid_U, \mathsf{pk}_U)$ from $\mathcal{F}_{\text{BB}}$ to O.
3. Upon receiving instructions from $\mathcal{Z}$ to send (com_{fp}, σ_{fp}) from O to U:
   - Check[a] that there exists an entry $(\cdot, com_{fp}, \sigma_{fp})$ in $f_{\text{FP}}(\texttt{ureg})$.
4. Upon receiving $(\texttt{ureg}, fp, decom_{fp}, \texttt{in}_O)$ from O to $\mathcal{F}_{\text{PPA}}$:
   - Check[a] that $(com_{fp}, \sigma_{fp}) \in f_{\text{FP}}(\texttt{ureg})$.
   - Check[a] that $\mathsf{COM.open}(com_{fp}, decom_{fp}, fp) = 1$.

   ↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{ureg}, fp, \texttt{in}_O)$ in the name of O.
5. Upon receiving leak ℓ from $\mathcal{F}_{\text{PUBA}}$ and output $(\texttt{out}_O)$ from $\mathcal{F}_{\text{PUBA}}$:
   - $(com_{\mathcal{UH}}, decom_{\mathcal{UH}}) \leftarrow \mathsf{commit}(0)$

   ↪ Report output $(com_{\mathcal{UH}}, \texttt{out}_O)$ from $\mathcal{F}_{\text{PPA}}$ to O.
6. Upon receiving instructions from $\mathcal{Z}$ to send $(com_{\mathcal{UH}}, ser^{(O)}, com^{(O)}_{ser}, decom^{(O)}_{ser}, com_{lin}, decom_{lin}, com_{id}, decom_{id}, \sigma)$ from O to U:
   - $com_{ser} := com^{(O)}_{ser} \oplus com^{(U)}_{ser}$
   - $decom_{ser} := decom^{(O)}_{ser} \oplus decom^{(U)}_{ser}$
   - $\mathsf{Log}^{\text{new}} := (0, com_{\mathcal{UH}}, decom_{\mathcal{UH}}, ser^{(O)}, com_{ser}, decom_{ser}, 0, com_{lin}, decom_{lin}, 0, com_{id}, decom_{id}, \sigma)$
   - $\mathsf{Call}^{a}\ \Pi_{\text{Verify}}(\mathsf{Log}^{\text{new}})$

   ↪ Allow $\mathcal{F}_{\text{PUBA}}$ to deliver output to U.

. . . . . .

U and O corrupted: Nothing to do.

$\{\Pi_{\text{PUBA}}\text{-}Bookkeeping^{(k)}\}_{k=1}^{K}$:

. . . . . .

U honest, O corrupted:

1. Upon receiving $(\texttt{bookkeep}k, \text{User})$ from $\mathcal{F}_{\text{PUBA}}$:
   - $ser \xleftarrow{r} \mathbb{Z}_p$
   - $(com^{(U)}_{ser^{\text{new}}}, decom^{(U)}_{ser^{\text{new}}}) \leftarrow \text{COM.commit}(\mathbf{0})$
   - $(\overline{com_{\mathcal{UH}}}, \overline{decom_{\mathcal{UH}}}) \leftarrow \text{COM.commit}(\mathbf{0})$
   - $(\overline{com_{lin}}, \overline{decom_{lin}}) \leftarrow \text{COM.commit}(\mathbf{0})$
   - $(\overline{com_{id}}, \overline{decom_{id}}) \leftarrow \text{COM.commit}(\mathbf{0})$
   - $stmt_{Val} \coloneqq (\overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \mathsf{vk}_O)$
   - $\Pi_{Val} \leftarrow \text{POK.SimProof}(td_{\text{sim}}, stmt_{Val}, \mathcal{R}_{Val})$

   $\hookrightarrow$ Report message $(\overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \Pi_{Val}, com^{(U)}_{ser^{\text{new}}})$ from U to O.
2. Upon receiving instructions from $\mathcal{Z}$ to send (com_{fp}, σ_{fp}) from O to U:
   - Check[a] that $(com_{fp}, \sigma_{fp}) \in f_{\text{FP}}(\texttt{ureg})$.
3. Upon receiving $(\texttt{bookkeep}k, \overline{com_{\mathcal{UH}}}', fp, decom_{fp}, \texttt{in}_O)$ from O to $\mathcal{F}_{\text{PPA}}$:
   - Check[a] $\overline{com_{\mathcal{UH}}}' = \overline{com_{\mathcal{UH}}}$
   - Check[a] $\text{COM.open}(com_{fp}, decom_{fp}, fp) = 1$.

   $\hookrightarrow$ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{bookkeep}k, fp, \texttt{in}_O)$ in the name of O.
4. Upon receiving leak ℓ from $\mathcal{F}_{\text{PUBA}}$ and output $(\texttt{out}_O, \alpha, \mathbf{s})$ to O from $\mathcal{F}_{\text{PUBA}}$:
   - $(com_{\mathsf{a}}, decom_{\mathsf{a}}) \leftarrow \text{COM.commit}(\mathbf{0})$
   - **if** $\alpha \neq \perp \vee \mathbf{s} \neq \perp$ **then**
     - $(com'_{\mathcal{UH}}, decom'_{\mathcal{UH}}) \leftarrow \text{COM.commit}(\mathbf{0})$
     - $(com''_{\mathcal{UH}}, decom''_{\mathcal{UH}}) \leftarrow \text{COM.commit}(\mathbf{0})$
     - $stmt_{Tr} \coloneqq (\overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \alpha, \mathbf{s})$
     - $\Pi_{Tr} \leftarrow \text{POK.SimProof}(td_{\text{sim}}, stmt_{Tr}, \mathcal{R}_{Tr})$
     - $decom^{\text{new}}_{\mathcal{UH}} \coloneqq decom''_{\mathcal{UH}} \oplus decom_{\mathsf{a}}$
   - **else**
     - $decom^{\text{new}}_{\mathcal{UH}} \coloneqq \overline{decom_{\mathcal{UH}}} \oplus decom_{\mathsf{a}}$

   $\hookrightarrow$ Report output $(\alpha, \mathbf{s}, com_{\mathsf{a}}, \texttt{out}_O)$ from $\mathcal{F}_{\text{PPA}}$ to O.
5. **if** $\alpha \neq \perp \vee \mathbf{s} \neq \perp$ **then**

   $\hookrightarrow$ Report message $(com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \Pi_{Tr})$ from U to O.
6. Upon receiving instructions from $\mathcal{Z}$ to send $(com^{\text{new}}_{\mathcal{UH}}, (ser^{\text{new}})^{(O)}, com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}, \sigma^{\text{new}})$ from O to U:
   - $ser^{\text{new}} \coloneqq (ser^{\text{new}})^{(O)} + (ser^{\text{new}})^{(U)}$
   - $com^{\text{new}}_{ser} \coloneqq com^{(O)}_{ser^{\text{new}}} \oplus com^{(U)}_{ser^{\text{new}}}$
   - $decom^{\text{new}}_{ser} \coloneqq decom^{(O)}_{ser^{\text{new}}} \oplus decom^{(U)}_{ser^{\text{new}}}$
   - $\text{Log}^{\text{new}} \coloneqq (0, com^{\text{new}}_{\mathcal{UH}}, decom^{\text{new}}_{\mathcal{UH}}, (ser^{\text{new}})^{(O)}, com^{\text{new}}_{ser}, decom^{\text{new}}_{ser}, 0, \overline{com_{lin}}, \overline{decom_{lin}}, 0, \overline{com_{id}}, \overline{decom_{id}}, \sigma^{\text{new}})$
   - Call[a] $\Pi_{\text{Verify}}(\text{Log}^{\text{new}})$

   $\hookrightarrow$ Allow $\mathcal{F}_{\text{PUBA}}$ to deliver output to U.

. . . . . .

U and O corrupted: Nothing to do but relay messages.

$\{\Pi_{\text{PUBA}}\text{-}Outsource^{(k)}\}^K_{k=1}$:

. . . . . .

U and P honest, O corrupted:
1. Upon receiving $(\texttt{outsource}k, \text{User})$ from $\mathcal{F}_{\text{PUBA}}$ and $(\texttt{outsource}k, pid_P)$ from $\mathcal{F}_{\text{PUBA}}$:
   - $ser \xleftarrow{r} \mathbb{Z}_p$
   - $(\overline{com_{id}}, \overline{decom_{id}}) \leftarrow \text{COM.commit}(\mathbf{0})$
   - $(\langle \mathcal{UH} \rangle^P, \langle \mathcal{UH} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \texttt{in}_U \rangle^P, \langle \texttt{in}_U \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \mathsf{o}_{\text{OUT}} \rangle^P, \langle \mathsf{o}_{\text{OUT}} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \mathsf{o}_{\text{UNV}} \rangle^P, \langle \mathsf{o}_{\text{UNV}} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \mathsf{o}_\alpha \rangle^P, \langle \mathsf{o}_\alpha \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \mathsf{o}_{\mathbf{s}} \rangle^P, \langle \mathsf{o}_{\mathbf{s}} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \mathsf{o}_{\mathsf{a}} \rangle^P, \langle \mathsf{o}_{\mathsf{a}} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - Parse $(\mathcal{UH}^{(P)}, com^{(O)}_{\mathcal{UH}}, decom^{(P)}_{\mathcal{UH}}) \coloneqq \langle \mathcal{UH} \rangle^P$
   - $stmt \coloneqq (\langle \mathcal{UH} \rangle^O, com^{(O)}_{\mathcal{UH}}, ser, \overline{com_{id}}, \mathsf{vk}_O)$
   - $\Pi \leftarrow \text{POK.SimProof}(td_{\text{sim}}, stmt, \mathcal{R}_{\mathsf{OS}})$
   - $(com^{(U)}_{ser^{\text{new}}}, decom^{(U)}_{ser^{\text{new}}}) \leftarrow \text{COM.commit}(\mathbf{0})$

↪ Report message $(\langle \mathcal{UH} \rangle^O, \langle \mathtt{in}_U \rangle^O, \langle \mathtt{o}_{\mathrm{OUT}} \rangle^O, \langle \mathtt{o}_{\mathrm{UNV}} \rangle^O, \langle \mathtt{o}_\alpha \rangle^O, \langle \mathtt{o}_\mathtt{s} \rangle^O, \langle \mathtt{o}_\mathtt{a} \rangle^O, com^{(O)}_{\mathcal{UH}}, ser, \overline{com_{id}}, \Pi, com^{(U)}_{ser^{\mathrm{new}}})$ from U to O.

2. Upon receiving instructions from $\mathcal{Z}$ to send $(com^{(O)}_{lin^{\mathrm{new}}}, com^{(P)}_{\mathcal{UH}}, com^{(P)}_{\mathtt{in}_U}, com^{(P)}_{\mathtt{o}_{\mathrm{OUT}}}, com^{(P)}_{\mathtt{o}_{\mathrm{UNV}}}, com^{(P)}_{\mathtt{o}_\alpha}, com^{(P)}_{\mathtt{o}_\mathtt{s}}, com^{(P)}_{\mathtt{o}_\mathtt{a}})$ from O to P:
   - Check[a] that all commitments are the ones sent earlier.
   - $(lin^{\mathrm{new}})^{(P)} \xleftarrow{r} \mathbb{Z}_p$

   Parse:
   - $\left(\mathcal{UH}^{(P)}, com^{(O)}_{\mathcal{UH}}, decom^{(P)}_{\mathcal{UH}}\right) \coloneqq \langle \mathcal{UH} \rangle^P$
   - $\left(\mathtt{in}^P_U, com^{(O)}_{\mathtt{in}_U}, decom^P_{\mathtt{in}_U}\right) \coloneqq \langle \mathtt{in}_U \rangle^P$
   - $\left(\mathtt{o}_{\mathrm{OUT}}{}^P, com^{(O)}_{\mathtt{o}_{\mathrm{OUT}}}, decom^P_{\mathtt{o}_{\mathrm{OUT}}}\right) \coloneqq \langle \mathtt{o}_{\mathrm{OUT}} \rangle^P$
   - $\left(\mathtt{o}_{\mathrm{UNV}}{}^P, com^{(O)}_{\mathtt{o}_{\mathrm{UNV}}}, decom^P_{\mathtt{o}_{\mathrm{UNV}}}\right) \coloneqq \langle \mathtt{o}_{\mathrm{UNV}} \rangle^P$
   - $\left(\mathtt{o}_\alpha{}^P, com^{(O)}_{\mathtt{o}_\alpha}, decom^P_{\mathtt{o}_\alpha}\right) \coloneqq \langle \mathtt{o}_\alpha \rangle^P$
   - $\left(\mathtt{o}_\mathtt{s}{}^P, com^{(O)}_{\mathtt{o}_\mathtt{s}}, decom^P_{\mathtt{o}_\mathtt{s}}\right) \coloneqq \langle \mathtt{o}_\mathtt{s} \rangle^P$
   - $\left(\mathtt{o}_\mathtt{a}{}^P, com^{(O)}_{\mathtt{o}_\mathtt{a}}, decom^P_{\mathtt{o}_\mathtt{a}}\right) \coloneqq \langle \mathtt{o}_\mathtt{a} \rangle^P$

   ↪ Report message $((lin^{\mathrm{new}})^{(P)}, com^{(O)}_{\mathcal{UH}}, com^{(O)}_{\mathtt{in}_U}, com^{(O)}_{\mathtt{o}_{\mathrm{OUT}}}, com^{(O)}_{\mathtt{o}_{\mathrm{UNV}}}, com^{(O)}_{\mathtt{o}_\alpha}, com^{(O)}_{\mathtt{o}_\mathtt{s}}, com^{(O)}_{\mathtt{o}_\mathtt{a}})^{(O)}$ from P to O.

3. Upon receiving instructions from $\mathcal{Z}$ to send $(lin^{\mathrm{new}})^{(O)}, decom^{(O)}_{lin^{\mathrm{new}}})$ from O to P:
   - Check[a] $\mathrm{COM.open}(com^{(O)}_{lin^{\mathrm{new}}}, decom^{(O)}_{lin^{\mathrm{new}}}, (lin^{\mathrm{new}})^{(O)}) \stackrel{?}{=} 1$
   - $lin^{\mathrm{new}} \coloneqq (lin^{\mathrm{new}})^{(P)} + (lin^{\mathrm{new}})^{(O)}$
   - Append $(lin^{\mathrm{new}}, \langle \mathcal{UH} \rangle^P, \langle \mathtt{in}_U \rangle^P, \langle \mathtt{o}_{\mathrm{OUT}} \rangle^P, \langle \mathtt{o}_{\mathrm{UNV}} \rangle^P, \langle \mathtt{o}_\alpha \rangle^P, \langle \mathtt{o}_\mathtt{s} \rangle^P, \langle \mathtt{o}_\mathtt{a} \rangle^P)$ to $f_{\mathrm{OI}}(pid_P, k)$
   - Load current subsession id $ssid$ and append $ssid \mapsto lin^{\mathrm{new}}$ to f_{LN}

4. Upon receiving instructions from $\mathcal{Z}$ to send $((ser^{\mathrm{new}})^{(O)}, com^{(O)}_{ser^{\mathrm{new}}}, decom^{(O)}_{ser^{\mathrm{new}}}, com^{\mathrm{new}}_{lin}, decom^{\mathrm{new}}_{lin}, \sigma^{\mathrm{new}})$ from O to U:
   - Parse $(\mathcal{UH}^{(O)}, com^{(P)}_{\mathcal{UH}}, decom^{(O)}_{\mathcal{UH}}) \coloneqq \langle \mathcal{UH} \rangle^O$
   - $com^{\mathrm{new}}_{\mathcal{UH}} \coloneqq com^{(P)}_{\mathcal{UH}} \oplus com^{(O)}_{\mathcal{UH}}$
   - $decom^{\mathrm{new}}_{\mathcal{UH}} \coloneqq decom^{(P)}_{\mathcal{UH}} \oplus decom^{(O)}_{\mathcal{UH}}$
   - $com^{\mathrm{new}}_{ser} \coloneqq com^{(U)}_{ser^{\mathrm{new}}} \oplus com^{(O)}_{ser^{\mathrm{new}}}$
   - $decom^{\mathrm{new}}_{ser} \coloneqq decom^{(U)}_{ser^{\mathrm{new}}} \oplus decom^{(O)}_{ser^{\mathrm{new}}}$
   - $\mathrm{Log}^{\mathrm{new}} \coloneqq (\mathbf{0}, com^{\mathrm{new}}_{\mathcal{UH}}, decom^{\mathrm{new}}_{\mathcal{UH}}, (ser^{\mathrm{new}})^{(O)}, com^{\mathrm{new}}_{ser}, decom^{\mathrm{new}}_{ser}, lin^{\mathrm{new}}, com^{\mathrm{new}}_{lin}, decom^{\mathrm{new}}_{lin}, \mathbf{0}, \overline{com_{id}}, \overline{decom_{id}}, \sigma^{\mathrm{new}})$
   - Call[a] $\Pi_{\mathrm{Verify}}(\mathrm{Log}^{\mathrm{new}})$

   ↪ Call ideal functionality $\mathcal{F}_{\mathrm{PUBA}}$ with input $(\mathtt{outsource}k)$ in the name of O.

5. Allow $\mathcal{F}_{\mathrm{PUBA}}$ to deliver output to all parties.

P honest, U and O corrupted:

1. Upon receiving $(\mathtt{outsource}k, pid_P)$ from $\mathcal{F}_{\mathrm{PUBA}}$ and instructions from $\mathcal{Z}$ to send $(\langle \mathcal{UH} \rangle^P, \langle \mathtt{in}_U \rangle^P, \langle \mathtt{o}_{\mathrm{OUT}} \rangle^P, \langle \mathtt{o}_{\mathrm{UNV}} \rangle^P, \langle \mathtt{o}_\alpha \rangle^P, \langle \mathtt{o}_\mathtt{s} \rangle^P, \langle \mathtt{o}_\mathtt{a} \rangle^P)$ from U to P and $(com^{(O)}_{lin^{\mathrm{new}}}, com^{(P)}_{\mathcal{UH}}, com^{(P)}_{\mathtt{in}_U}, com^{(P)}_{\mathtt{o}_{\mathrm{OUT}}}, com^{(P)}_{\mathtt{o}_{\mathrm{UNV}}}, com^{(P)}_{\mathtt{o}_\alpha}, com^{(P)}_{\mathtt{o}_\mathtt{s}}, com^{(P)}_{\mathtt{o}_\mathtt{a}})$ from O to P:

   Parse:
   - $\left(\mathcal{UH}^{(P)}, com^{(O)}_{\mathcal{UH}}, decom^{(P)}_{\mathcal{UH}}\right) \coloneqq \langle \mathcal{UH} \rangle^P$
   - $\left(\mathtt{in}^P_U, com^{(O)}_{\mathtt{in}_U}, decom^P_{\mathtt{in}_U}\right) \coloneqq \langle \mathtt{in}_U \rangle^P$
   - $\left(\mathtt{o}_{\mathrm{OUT}}{}^P, com^{(O)}_{\mathtt{o}_{\mathrm{OUT}}}, decom^P_{\mathtt{o}_{\mathrm{OUT}}}\right) \coloneqq \langle \mathtt{o}_{\mathrm{OUT}} \rangle^P$
   - $\left(\mathtt{o}_{\mathrm{UNV}}{}^P, com^{(O)}_{\mathtt{o}_{\mathrm{UNV}}}, decom^P_{\mathtt{o}_{\mathrm{UNV}}}\right) \coloneqq \langle \mathtt{o}_{\mathrm{UNV}} \rangle^P$
   - $\left(\mathtt{o}_\alpha{}^P, com^{(O)}_{\mathtt{o}_\alpha}, decom^P_{\mathtt{o}_\alpha}\right) \coloneqq \langle \mathtt{o}_\alpha \rangle^P$
   - $\left(\mathtt{o}_\mathtt{s}{}^P, com^{(O)}_{\mathtt{o}_\mathtt{s}}, decom^P_{\mathtt{o}_\mathtt{s}}\right) \coloneqq \langle \mathtt{o}_\mathtt{s} \rangle^P$
   - $\left(\mathtt{o}_\mathtt{a}{}^P, com^{(O)}_{\mathtt{o}_\mathtt{a}}, decom^P_{\mathtt{o}_\mathtt{a}}\right) \coloneqq \langle \mathtt{o}_\mathtt{a} \rangle^P$

   Check shares
   - Check[a] $\mathrm{open}(com^{(P)}_{\mathcal{UH}}, decom^{(P)}_{\mathcal{UH}}, \mathcal{UH}^{(P)}) \stackrel{?}{=} 1$
   - Check[a] $\mathrm{open}(com^{(P)}_{\mathtt{in}_U}, decom^P_{\mathtt{in}_U}, \mathtt{in}^P_U) \stackrel{?}{=} 1$
   - Check[a] $\mathrm{open}(com^{(P)}_{\mathtt{o}_{\mathrm{OUT}}}, decom^P_{\mathtt{o}_{\mathrm{OUT}}}, \mathtt{o}_{\mathrm{OUT}}{}^P) \stackrel{?}{=} 1$

- Check[a] open$(com^{(P)}_{\mathrm{o_{UNV}}}, decom^{P}_{\mathrm{o_{UNV}}}, \mathrm{o_{UNV}}^{P}) \stackrel{?}{=} 1$
- Check[a] open$(com^{(P)}_{\mathrm{o}_\alpha}, decom^{P}_{\mathrm{o}_\alpha}, \mathrm{o}_\alpha{}^{P}) \stackrel{?}{=} 1$
- Check[a] open$(com^{(P)}_{\mathrm{o_s}}, decom^{P}_{\mathrm{o_s}}, \mathrm{o_s}^{P}) \stackrel{?}{=} 1$
- Check[a] open$(com^{(P)}_{\mathrm{o_a}}, decom^{P}_{\mathrm{o_a}}, \mathrm{o_a}^{P}) \stackrel{?}{=} 1$

Perform coin toss with O

- $(lin^{\mathrm{new}})^{(P)} \xleftarrow{r} \mathbb{Z}_p$

↪ Report message $((lin^{\mathrm{new}})^{(P)},\ com^{(O)}_{\mathcal{UH}},\ com^{(O)}_{\mathtt{in}_U},\ com^{(O)}_{\mathrm{o_{OUT}}},\ com^{(O)}_{\mathrm{o_{UNV}}},\ com^{(O)}_{\mathrm{o}_\alpha},\ com^{(O)}_{\mathrm{o_s}},\ com^{(O)}_{\mathrm{o_a}})$ from P to O.

2. Upon receiving instructions from $\mathcal{Z}$ to send $((lin^{\mathrm{new}})^{(O)},\ decom^{(O)}_{lin^{\mathrm{new}}})$ from O to P:
   - Check[a] COM.open$(com^{(O)}_{lin^{\mathrm{new}}},\ decom^{(O)}_{lin^{\mathrm{new}}},\ (lin^{\mathrm{new}})^{(O)}) \stackrel{?}{=} 1$
   - $lin^{\mathrm{new}} \coloneqq (lin^{\mathrm{new}})^{(P)} + (lin^{\mathrm{new}})^{(O)}$
   - Append $(lin^{\mathrm{new}}, \langle\mathcal{UH}\rangle^P, \langle\mathtt{in}_U\rangle^P, \langle\mathrm{o_{OUT}}\rangle^P, \langle\mathrm{o_{UNV}}\rangle^P, \langle\mathrm{o}_\alpha\rangle^P, \langle\mathrm{o_s}\rangle^P, \langle\mathrm{o_a}\rangle^P)$ to $f_{\mathrm{OI}}(pid_P, k)$
   - Load current subsession id *ssid* and append $ssid \mapsto lin^{\mathrm{new}}$ to f_{LN}.

   ↪ Report message (lin^{new}) from P to U.
   ↪ Call ideal functionality $\mathcal{F}_{\mathrm{PUBA}}$ with input $(\mathtt{outsource}k, \bot)$ in the name of U.
   ↪ Call ideal functionality $\mathcal{F}_{\mathrm{PUBA}}$ with input $(\mathtt{outsource}k)$ in the name of O.
3. Allow $\mathcal{F}_{\mathrm{PUBA}}$ to deliver output to all parties.

$\{\Pi_{\mathrm{PUBA}}\text{-}Analytics^{(k)}\}^{K}_{k=1}$:

P honest, O corrupted:

1. Upon receiving instructions from $\mathcal{Z}$ to send $(com_{fp},\ \sigma_{fp})$ from O to U:
   - Check[a] that $(com_{fp},\ \sigma_{fp}) \in f_{\mathrm{FP}}(\mathtt{analyt}k)$.
2. Upon receiving $(\mathtt{analyt}k, pid_P)$ from $\mathcal{F}_{\mathrm{PUBA}}$ and $(\mathtt{analyt}k,\ \{(\langle\mathcal{UH}\rangle^O, \langle\mathtt{in}\rangle^O, \langle\mathrm{o_{OUT}}\rangle^O, \langle\mathrm{o_{UNV}}\rangle^O, \langle\mathrm{o}_\alpha\rangle^O, \langle\mathrm{o_s}\rangle^O, \langle\mathrm{o_a}\rangle^O)_z\}^{Z_k}_{z=1},\ fp,\ decom_{fp},\ \mathtt{in}_O)$ from O to $\mathcal{F}_{\mathrm{PPA}}$:
   - Check[a] that COM.open$(com_{fp},\ decom_{fp},\ fp) = 1$.
   - Load[a] and remove the first Z_k entries $\{(lin,\ \langle\mathcal{UH}\rangle^P,\ \langle\mathtt{in}\rangle^P,\ \langle\mathrm{o_{OUT}}\rangle^P,\ \langle\mathrm{o_{UNV}}\rangle^P,\ \langle\mathrm{o}_\alpha\rangle^P,\ \langle\mathrm{o_s}\rangle^P,\ \langle\mathrm{o_a}\rangle^P)_z\}^{Z_k}_{z=1}$ from $f_{\mathrm{OI}}(pid_P,\ k)$
   - For every z from 1 to Z, combine[a]:
     - $\mathcal{UH}_z \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle\mathcal{UH}_z\rangle^P, \langle\mathcal{UH}_z\rangle^O)$
     - $\mathtt{in}_z \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle\mathtt{in}_z\rangle^P, \langle\mathtt{in}_z\rangle^O)$
     - $\mathrm{o_{OUT}}_z \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle\mathrm{o_{OUT}}_z\rangle^P, \langle\mathrm{o_{OUT}}_z\rangle^O)$
     - $\mathrm{o_{UNV}}_z \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle\mathrm{o_{UNV}}_z\rangle^P, \langle\mathrm{o_{UNV}}_z\rangle^O)$
     - $\mathrm{o}_{\alpha z} \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle\mathrm{o}_{\alpha z}\rangle^P, \langle\mathrm{o}_{\alpha z}\rangle^O)$
     - $\mathrm{o_s}_z \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle\mathrm{o_s}_z\rangle^P, \langle\mathrm{o_s}_z\rangle^O)$
     - $\mathrm{o_a}_z \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle\mathrm{o_a}_z\rangle^P, \langle\mathrm{o_a}_z\rangle^O)$

   ↪ Call ideal functionality $\mathcal{F}_{\mathrm{PUBA}}$ with input $(\mathtt{analyt}k,\ fp,\ \mathtt{in}_O)$ in the name of O and receive leak ℓ.
3. Upon being asked by $\mathcal{F}_{\mathrm{PUBA}}$ for updated inputs for $(z_1, \ldots, z_n)$:
   ↪ Provide inputs $\{(\mathcal{UH}_{z_i},\ \mathtt{in}_{z_i}) \mid 1 \le i \le n\}$ to ideal functionality $\mathcal{F}_{\mathrm{PUBA}}$.
4. Upon receiving leak $\{(z_i,\ \alpha_z,\ \mathbf{s}_z,\ \mathbf{a}_z,\ \mathtt{out}_{z_i}) \mid 1 \le i \le n\}$ from $\mathcal{F}_{\mathrm{PUBA}}$:
   - For every z_i with $1 \le i \le n$:
     - $(com_{\mathbf{a}_{z_i}}, decom_{\mathbf{a}_z}) \xleftarrow{r} \mathrm{commit}(\mathbf{a}_z)$
     - $ct_{\mathrm{out}_z} \coloneqq \mathtt{out}_{z_i} + \mathrm{o_{OUT}}_z$
   - For all other $z \le Z_k$:
     - $(com_{\mathbf{a}_{z_i}}, decom_{\mathbf{a}_z}) \xleftarrow{r} \mathrm{commit}(0)$
     - $r \xleftarrow{r} \mathcal{IN}$
     - $ct_{\mathrm{out}_z} \coloneqq r + \mathrm{o_{OUT}}_z$
     - $\mathbf{a}_z = \bot$

   ↪ Allow $\mathcal{F}_{\mathrm{PUBA}}$ to continue.
5. Upon receiving output $(\{\alpha_z, \mathbf{s}_z\}^{Z_k}_{z=1},\ \mathtt{out}_O)$ from $\mathcal{F}_{\mathrm{PUBA}}$ to O:
   - For each $z \in 1 \ldots Z$:
     - $f_{\mathrm{UI}}(lin_z) \coloneqq (com_{\mathbf{a}_z},\ decom_{\mathbf{a}_z},\ \mathbf{a}_z,\ \alpha_z,\ \mathbf{s}_z,\ \mathrm{o}_{\alpha z},\ \mathrm{o_s}_z,\ \mathrm{o_a}_z,\ \mathrm{o_{UNV}},\ ct_{\mathrm{out}_z})$

   ↪ Report output $(\{(\alpha_z,\ \mathbf{s}_z,\ com_{\mathbf{a}_z},\ ct_{\mathrm{out}_z}) \mid 1 \le z \le Z_k\},\ \mathtt{out}_O)$ from $\mathcal{F}_{\mathrm{PPA}}$ to O.

Π_{PUBA}-*Update*:

U and P honest, O corrupted:

1. Upon receiving $(\texttt{update}, \text{User})$ from $\mathcal{F}_{\text{PUBA}}$ and $(\texttt{update}, pid_P)$ from $\mathcal{F}_{\text{PUBA}}$:
   ↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{update})$ in the name of O.
2. Upon receiving leak $ssid$ from $\mathcal{F}_{\text{PUBA}}$:
   - Load $lin \coloneqq f_{\text{LN}}(ssid)$
   - Load[a] and remove $(com'_{\mathbf{a}}, decom'_{\mathbf{a}}, \mathbf{a}', \alpha', \mathbf{s}', \mathrm{o}_\alpha, \mathrm{o}_{\mathbf{s}}, \mathrm{o}_{\mathbf{a}}, \mathrm{o}_{\text{UNV}}, ct_{\texttt{out}_U}{}') \coloneqq f_{\text{UI}}(lin)$
   - $(\overline{com_{\mathcal{UH}}}, \overline{decom_{\mathcal{UH}}}) \leftarrow \text{COM.commit}(\mathbf{0})$
   - $(\overline{com_{id}}, decom_{id}) \leftarrow \text{COM.commit}(\mathbf{0})$
   - $ser \xleftarrow{\text{r}} \mathbb{Z}_p$
   - $stmt \coloneqq (\overline{com_{\mathcal{UH}}}, ser, lin, \overline{com_{id}}, \mathsf{vk}_O)$
   - $\Pi \leftarrow \text{POK.SimProof}(td_{\text{sim}}, stmt, \mathcal{R}_{Upd})$
   - $(com^{(U)}_{ser^{\text{new}}}, decom^{(U)}_{ser^{\text{new}}}) \leftarrow \text{COM.commit}(\mathbf{0})$
3. **if** $\alpha \neq \perp \wedge \mathbf{s} \neq \perp$ **then**
   - $(com'_{\mathcal{UH}}, decom'_{\mathcal{UH}}) \leftarrow \text{COM.commit}(\mathbf{0})$
   - $(com''_{\mathcal{UH}}, decom''_{\mathcal{UH}}) \leftarrow \text{COM.commit}(\mathbf{0})$
   - $decom^{\text{new}}_{\mathcal{UH}} \coloneqq \overline{decom''_{\mathcal{UH}}} \oplus decom_{\mathbf{a}}$
   - $stmt_{Tr} \coloneqq (\overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \alpha, \mathbf{s})$
   - $\Pi_{Tr} \leftarrow \text{POK.SimProof}(td_{\text{sim}}, stmt_{Tr}, \mathcal{R}_{Tr})$
     ↪ Report message $(\overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \overline{com_{id}}, ser, lin, \Pi, \Pi_{Tr}, com^{(U)}_{ser})$ from U to O.
4. **else**
   - $decom^{\text{new}}_{\mathcal{UH}} \coloneqq \overline{decom_{\mathcal{UH}}} \oplus decom_{\mathbf{a}}$
     ↪ Report message $(\overline{com_{\mathcal{UH}}}, \overline{com_{id}}, ser, lin, \Pi, com^{(U)}_{ser})$ from U to O.
5. **fi**
   ↪ Allow $\mathcal{F}_{\text{PUBA}}$ to continue.
6. Upon receiving instructions from $\mathcal{Z}$ to send $(com_{\mathbf{a}}, com^{\text{new}}_{\mathcal{UH}}, (ser^{\text{new}})^{(O)}, com^{(O)}_{ser^{\text{new}}}, decom^{(O)}_{ser^{\text{new}}}, com^{\text{new}}_{lin}, decom^{\text{new}}_{lin}, \sigma^{\text{new}}, ct_{\texttt{out}_U})$ from O to U:
   - Check[a] $com_{\mathbf{a}} \stackrel{?}{=} com'_{\mathbf{a}}$
   - Check[a] $ct_{\texttt{out}_U} \stackrel{?}{=} ct_{\texttt{out}_U}{}'$
   - $com^{\text{new}}_{ser} \coloneqq com^{(O)}_{ser^{\text{new}}} \oplus com^{(U)}_{ser^{\text{new}}}$
   - $decom^{\text{new}}_{ser} \coloneqq decom^{(O)}_{ser^{\text{new}}} \oplus decom^{(U)}_{ser^{\text{new}}}$
   - $\text{Log}^{\text{new}} \coloneqq (0, com^{\text{new}}_{\mathcal{UH}}, decom^{\text{new}}_{\mathcal{UH}}, (ser^{\text{new}})^{(O)}, com^{\text{new}}_{ser}, decom^{\text{new}}_{ser}, 0, com^{\text{new}}_{lin}, decom^{\text{new}}_{lin}, \mathbf{0}, \overline{com_{id}}, \overline{decom_{id}}, \sigma^{\text{new}})$
   - Call[a] $\Pi_{\text{Verify}}(\text{Log}^{\text{new}})$
   ↪ Allow $\mathcal{F}_{\text{PUBA}}$ to deliver output to all parties.

P honest, U and O corrupted:

1. Upon receiving $(\texttt{update}, pid_P)$ from $\mathcal{F}_{\text{PUBA}}$ and instructions from $\mathcal{Z}$ to send lin from U to P:
   - Load[a] and remove $(com_{\mathbf{a}}, decom_{\mathbf{a}}, \mathbf{a}, \alpha, \mathbf{s}, \mathrm{o}_\alpha, \mathrm{o}_{\mathbf{s}}, \mathrm{o}_{\mathbf{a}}, \mathrm{o}_{\text{UNV}}, ct_{\texttt{out}_U}) \coloneqq f_{\text{UI}}(lin)$
   - $ct_\alpha \coloneqq \alpha + \mathrm{o}_\alpha$
   - $ct_{\mathbf{s}} \coloneqq \mathbf{s} + \mathrm{o}_{\mathbf{s}}$
   - $ct_{\mathbf{a}} \coloneqq \mathbf{a} + \mathrm{o}_{\mathbf{a}}$
   - $ct_{decom_{\mathbf{a}}} \coloneqq decom_{\mathbf{a}} + \mathrm{o}_{\text{UNV}}$
   - $ct_{\texttt{out}_U} \coloneqq \texttt{out}_U + \mathrm{o}_{\text{OUT}}$
   ↪ Report message $(ct_\alpha, ct_{\mathbf{s}}, ct_{\mathbf{a}}, ct_{decom_{\mathbf{a}}}, ct_{\texttt{out}_U})$ from P to U.
   ↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{update})$ in the name of O and a (random) corrupted U.
2. Receive output $(\texttt{ok})$ from $\mathcal{F}_{\text{PUBA}}$ to U and O

[a] If this fails, output $\perp$ and abort.

We now introduce a series of hybrid games H_i and corresponding simulators $\mathcal{S}_i$ for protocols $\Pi_{\text{PUBA}i}$. Formally, given security parameter λ, each hybrid has the following form:

$$\mathsf{H}_i \coloneqq \text{EXEC}_{\Pi_{\text{PUBA}i},\mathcal{F}_{\text{BB}},\mathcal{F}_{\text{PPA}},\mathcal{S}_i,\mathcal{Z}}(1^\lambda)$$

We then show for each pair of consecutive hybrids H_i and H_{i+1}, that, given our underlying assumptions, no distinguisher can distinguish the two games better than by guessing.

For our proof, we consider the following hybrid games H_i:

H_1 The hybrid H_1 is equivalent to the real experiment. That is,

$$\mathsf{H}_1 \coloneqq \text{EXEC}_{\Pi_{\text{PUBA}},\mathcal{F}_{\text{BB}},\mathcal{F}_{\text{PPA}},\mathcal{S}_1,\mathcal{Z}}(1^\lambda)$$

This means that all parties execute the real protocol.

H_2 All calls to hybrid functionalities, namely to $\mathcal{F}_{\text{PPA}}$, $\mathcal{F}_{\text{KE}}$ and $\mathcal{F}_{\text{BB}}$, are replaced by calls to $\mathcal{S}_2$, who simulates their behavior.

H_3 Introduces a map f_{UI} to be used by the simulator $\mathcal{S}_3$, which is similar to what an honest proxy would store for the updates of the user history after an Analytics. During simulation of $\mathcal{F}_{\text{PPA}}$ for $\texttt{analyt}k$, i.e. after O sent a message $(\texttt{analyt}k, fp, decom_{fp}, [\{(\langle\mathcal{UH}\rangle^O, \langle\texttt{in}_U\rangle^O, \langle\text{o}_{\text{OUT}}\rangle^O, \langle\text{o}_{\text{UNV}}\rangle^O, \langle\text{o}_\alpha\rangle^O, \langle\text{o}_{\mathbf{s}}\rangle^O, \langle\text{o}_{\mathbf{a}}\rangle^O)_z\}_{z=1}^{Z_k}], \texttt{in}_O)$ to $\mathcal{F}_{\text{PPA}}$, and P has sent a message $[(\texttt{analyt}k, com_{fp}, \{(\langle\mathcal{UH}\rangle^P, \langle\texttt{in}_U\rangle^P, \langle\text{o}_{\text{OUT}}\rangle^P, \langle\text{o}_{\text{UNV}}\rangle^P, \langle\text{o}_\alpha\rangle^P, \langle\text{o}_{\mathbf{s}}\rangle^P, \langle\text{o}_{\mathbf{a}}\rangle^P)_z\}_{z=1}^{Z_k})]$ to $\mathcal{F}_{\text{PPA}}$, $\mathcal{S}_3$ computes Δ honestly (with fresh coins, if necessary), based on the two inputs. $\mathcal{S}_3$ uses Π_{Share}-*Combine* on the inputs to reconstruct $\mathcal{UH}_z$, $\texttt{in}_z$, $\text{o}_{\text{OUT}z}$ $\text{o}_{\text{UNV}z}$, $\text{o}_{\alpha z}$, $\text{o}_{\mathbf{s}z}$ and $\text{o}_{\mathbf{a}z}$ for each user $z \in [Z]$. If reconstruction on any of the shares fails, $\mathcal{S}_3$ aborts. With this, $\mathcal{S}_3$ computes $(com_{\mathbf{a}_z}, decom_{\mathbf{a}_z}) \leftarrow \text{commit}(\mathbf{a}_z)$ and $ct_{\texttt{out}_z} \coloneqq \texttt{out}_z + \text{o}_{\text{OUT}z}$. $\mathcal{S}_3$ then adds a new entry $(lin_z \mapsto (com_{\mathbf{a}_z}, decom_{\mathbf{a}_z}, \mathbf{a}_z, \alpha_z, \mathbf{s}_z, \text{o}_{\alpha z}, \text{o}_{\mathbf{s}z}, \text{o}_{\mathbf{a}z}, \text{o}_{\text{UNV}z}, ct_{\texttt{out}_z}))$ to f_{UI}.

H_4 During setup, instead of honestly sampling a *crs*, $\mathcal{S}_4$ computes $(crs_{\text{ZK}}, td_{\text{sim}}) \leftarrow \text{SetupSPoK}$ and publishes $crs \coloneqq crs_{\text{ZK}}$ as common reference string. Also, the simulator stores the verification key vk_O of the operator O, which is obtained by simulating $\mathcal{F}_{\text{BB}}$ during the initialization. The simulator stores td_{sim}, the remainder stays as it is.

H_5 Replaces all zero-knowledge proofs of honest parties by simulated proofs (using td_{sim}) created by the simulator. Note that the simulated proofs can be created *independently* from (thus without knowing) the actual witness.

H_6 During simulation of the Bookkeeping task, instead of computing the addition vector $\mathbf{a}$ alongside its commitment and decommitment information honestly according to the function Δ the simulator uses $\mathbf{a} = \mathbf{0}$ alongside commitment- and decommitment-information $(com_{\mathbf{a}}, decom_{\mathbf{a}}) \leftarrow \text{COM.commit}(0)$. Note that the values for $\mathbf{a}$ and $decom_{\mathbf{a}}$ are not needed for simulation since H_5 so the only remaining value visible to the environment is $com_{\mathbf{a}}$.

H_7 All commitments that honest players create in the real protocol (i.e. those on *lin*, *ser*, $\mathcal{UH}$ and *id*) are now created by the simulator as $com_{\mathbf{0}}$, i.e. commitments to the zero-vector of appropriate size. Also, whenever the user U is supposed to send a serial number, $\mathcal{S}_7$ samples a new value $ser \xleftarrow{r} \mathbb{Z}_p$ and sends this instead of a real serial number.

H_8 All proxies P are replaced by an equivalent machine P', which behaves similar to P, with the exception that during the Outsource-task, P' sends to the simulator the shares $(lin, \langle\mathcal{UH}\rangle^P, \langle\mathtt{in}_U\rangle^P, \langle\mathsf{o}_{\mathrm{OUT}}\rangle^P, \langle\mathsf{o}_{\mathrm{UNV}}\rangle^P, \langle\mathsf{o}_\alpha\rangle^P, \langle\mathsf{o}_\mathbf{s}\rangle^P, \langle\mathsf{o}_\mathbf{a}\rangle^P)$.

H_9 Introduces a map f_{OI} to be used by the simulator $\mathcal{S}_9$, which is similar to what an honest proxy would store for the outsource information. It maps $pid_P \times [K]$ to a list $f_{\mathrm{OI}}(pid_P, k)$ of entries $(lin, \langle\mathcal{UH}\rangle^P, \langle\mathtt{in}_U\rangle^P, \langle\mathsf{o}_{\mathrm{OUT}}\rangle^P, \langle\mathsf{o}_{\mathrm{UNV}}\rangle^P, \langle\mathsf{o}_\alpha\rangle^P, \langle\mathsf{o}_\mathbf{s}\rangle^P, \langle\mathsf{o}_\mathbf{a}\rangle^P)$.
The map is only updated during Outsource-tasks. After $\mathcal{S}_9$ received the leak $(lin, \langle\mathcal{UH}\rangle^P, \langle\mathtt{in}_U\rangle^P, \langle\mathsf{o}_{\mathrm{OUT}}\rangle^P, \langle\mathsf{o}_{\mathrm{UNV}}\rangle^P, \langle\mathsf{o}_\alpha\rangle^P, \langle\mathsf{o}_\mathbf{s}\rangle^P, \langle\mathsf{o}_\mathbf{a}\rangle^P)$ from P', $\mathcal{S}_9$ adds an entry $(pid_P, k) \mapsto (lin, \langle\mathcal{UH}\rangle^P, \langle\mathtt{in}_U\rangle^P, \langle\mathsf{o}_{\mathrm{OUT}}\rangle^P, \langle\mathsf{o}_{\mathrm{UNV}}\rangle^P, \langle\mathsf{o}_\alpha\rangle^P, \langle\mathsf{o}_\mathbf{s}\rangle^P, \langle\mathsf{o}_\mathbf{a}\rangle^P)$ to f_{OI}.

H_{10} Introduces a map f_{LN} for the simulator which contains a mapping from the current subsession id *ssid* to the linking number *lin*. The map is only updated during Outsource-tasks. After receiving the leak $(lin, \langle\mathcal{UH}\rangle^P, \langle\mathtt{in}_U\rangle^P, \langle\mathsf{o}_{\mathrm{OUT}}\rangle^P, \langle\mathsf{o}_{\mathrm{UNV}}\rangle^P, \langle\mathsf{o}_\alpha\rangle^P, \langle\mathsf{o}_\mathbf{s}\rangle^P, \langle\mathsf{o}_\mathbf{a}\rangle^P)$ from the proxy P', $\mathcal{S}_{10}$ loads the current subsession id *ssid* and adds an entry $(ssid \to lin)$ to f_{LN}.

H_{11} All calls from honest parties of the form Π_{Share}-$Share(x)$ for an arbitrary $x \in \mathbb{Z}_p^n$ during the protocol execution are replaced by calls Π_{Share}-$Share(\mathbf{0})$ from the simulator, where $\mathbf{0} = 0^n$ is the all-zero vector of appropriate size. Furthermore, the simulator takes the role of the proxy P during the computation of the linking number during Outsource.
Thus the proxy no longer leaks $(lin, \langle\mathcal{UH}\rangle^P, \langle\mathtt{in}_U\rangle^P, \langle\mathsf{o}_{\mathrm{OUT}}\rangle^P, \langle\mathsf{o}_{\mathrm{UNV}}\rangle^P, \langle\mathsf{o}_\alpha\rangle^P, \langle\mathsf{o}_\mathbf{s}\rangle^P, \langle\mathsf{o}_\mathbf{a}\rangle^P)$ to $\mathcal{S}_{11}$.
During simulation of Outsource-tasks, linking numbers were created by the simulator and the operator. The simulator $\mathcal{S}_{11}$ still stores $(lin, \langle\mathcal{UH}\rangle^P, \langle\mathtt{in}_U\rangle^P, \langle\mathsf{o}_{\mathrm{OUT}}\rangle^P, \langle\mathsf{o}_{\mathrm{UNV}}\rangle^P, \langle\mathsf{o}_\alpha\rangle^P, \langle\mathsf{o}_\mathbf{s}\rangle^P, \langle\mathsf{o}_\mathbf{a}\rangle^P)$ in $f_{\mathrm{OI}}(pid_P, k)$.
The linking number *lin* is known to $\mathcal{S}_{11}$ due to its participation in the Blum-type coin toss. The shares are known regardless of the user's corruption. If the user is honest, the shares are created by $\mathcal{S}_{11}$ in the first place and can be stored directly. If the user is corrupted, the environment sends the shares to the proxy in the name of the user. This message is visible to the simulator.

H_{12} All honest user U are replaced by machines U' that run a similar code as U with the one exception that during Update-tasks, the actions are restricted to sending the linking number *lin* to the simulator $\mathcal{S}_{12}$.
The remaining part of the honest user's protocol for the Update task is played by the simulator.

In the honest user case, $\mathcal{S}_{12}$ fetches $(com_{\mathbf{a}_z},\ decom_{\mathbf{a}_z},\ \mathbf{a}_z,\ \alpha_z,\ \mathbf{s}_z,\ \mathrm{o}_{\alpha z},\ \mathrm{o}_{\mathbf{s}z},\ \mathrm{o}_{\mathbf{a}z},\ \mathrm{o}_{\mathrm{UNV}z},\ ct_{\mathtt{out}_z})$ from $f_{\mathrm{UI}}(lin)$ and follows the honest protocol of U from H_{11}.

H_{13} Introduces incorruptible entity $\mathcal{F}_{\mathrm{PUBA}}$ that follows the specification from Section A.3 into the experiment, which is only accessible by honest users and the simulator through subroutine input/output tapes.

H_{14} Replaces the trusted signing authority $\mathcal{TSA}$ with a dummy party that immediately forwards its input to the ideal functionality $\mathcal{F}_{\mathrm{PUBA}}$ and leaks $\mathtt{in}_O$ to $\mathcal{S}_{14}$. All interactions of $\mathcal{TSA}$ are simulated by $\mathcal{S}_{14}$ by following the original protocol.

H_{15} Introduces a map f_{FP} to be used by the simulator $\mathcal{S}_{15}$, which maps a task $\mathtt{task} \in \{\mathtt{ureg}, \mathtt{bookkeep}K, \mathtt{analyt}K\}$ to a set of tuples $(fp,\ com_{fp},\ \sigma_{fp})$ of FP and corresponding commitment and signature. During simulation of the task SignFP, in case an input fp can be used for task $\mathtt{task}$, the simulator computes the signature σ_{fp} on $(\mathtt{task},\ com_{fp})$ and stores $(com_{fp},\ \sigma_{fp})$ in $f_{\mathrm{FP}}(\mathtt{task})$.

The user U is replaced by a new user U', which skips verification of the signature on $(\mathtt{ureg},\ com_{fp})$ and instead asks the simulator if the certificate is valid. Instead of manually verifying the signature, $\mathcal{S}_{15}$ verifies that $(com_{fp},\ \sigma_{fp}) \in f_{\mathrm{FP}}(\mathtt{task})$.

H_{16} During simulation of $\mathcal{F}_{\mathrm{BB}}$ for the Init-Task, if the simulator receives a message $(\mathtt{Register}, pid_O, \mathsf{vk}_O)$ from O to $\mathcal{F}_{\mathrm{BB}}$, it follows the simulation procedure of $\mathcal{F}_{\mathrm{BB}}$ correctly and, if it succeeded, calls $\mathcal{F}_{\mathrm{PUBA}}$ with input $(\mathtt{init})$ in the name of O.

H_{17} Replaces all honest user U^* by dummy parties that immediately forward their input to the ideal functionality $\mathcal{F}_{\mathrm{PUBA}}$ with the additional property, that they still leak the linking number lin during the Update-task (see H_{12}) and still call $\mathcal{F}_{\mathrm{PPA}}$ with honest inputs when demanded by the protocol; the remaining protocol parts are executed just as specified in the protocol by the simulator $\mathcal{S}_{17}$.

$\mathcal{S}_{17}$ also controls input to $\mathcal{F}_{\mathrm{PUBA}}$ for corrupted users during simulation of Outsource- and Update-tasks. During the Outsource-task, $\mathcal{S}_{17}$ calls $\mathcal{F}_{\mathrm{PUBA}}$ in the name of U with empty input after the successful exchange of the linking number lin. In the Update-task, $\mathcal{S}_{17}$ calls $\mathcal{F}_{\mathrm{PUBA}}$ in the name of U with input $(\mathtt{update})$ after receiving the first message.

H_{18} Replaces the proxies P with dummy parties that forward their input to the ideal functionality $\mathcal{F}_{\mathrm{PUBA}}$. The remaining messages that the proxies sent will be simulated by the simulator by honestly following the protocol of P from H_{17}.

H_{19} The simulator now enforces that for every task, $\mathcal{F}_{\mathrm{PUBA}}$ is called by the O with the correct inputs. This causes $\mathcal{F}_{\mathrm{PUBA}}$ to have input from all parties, which means that it behaves according to its definition and provides leaks and output to $\mathcal{S}_{19}$, which the simulator can use. Thus, $\mathcal{S}_{19}$ can stop simulating $\mathcal{F}_{\mathrm{PPA}}$ by executing Δ and the appropriate simulator and instead uses the inputs received from O to $\mathcal{F}_{\mathrm{PPA}}$ in order to call $\mathcal{F}_{\mathrm{PUBA}}$ in the name of O. The leaks obtained by $\mathcal{F}_{\mathrm{PUBA}}$ are then used as output of $\mathcal{F}_{\mathrm{PPA}}$.

When executing Π_{Share}-*Combine* with an honest user and executing the Analytics-task with arbitrary user corruption, the simulation of $\mathcal{F}_{\text{PPA}}$ also includes a consistency check for the operator's input: If reconstruction with share via Π_{Share}-*Combine* fails, the simulator aborts. The operator shares are obtained via input to $\mathcal{F}_{\text{PPA}}$, the proxy shares are fetched from $f_{\text{OI}}(pid_P, k)$.

When $\mathcal{F}_{\text{PUBA}}$ asks $\mathcal{S}_{19}$ for updated inputs $(\mathcal{UH}_z, \texttt{in}_z)$ for corrupted users U_z during the Analytics-task, $\mathcal{S}_{19}$ uses Π_{Share}-*Combine* to reconstruct those values using the shares that U created during Outsource. The proxy-shares, $(\langle \mathcal{UH} \rangle^P, \langle \texttt{in} \rangle^P)$, are taken from $f_{\text{OI}}(pid_P, k)$. The operator-shares $\langle \mathcal{UH} \rangle^O$ and $\langle \texttt{in} \rangle^O$ are extracted from the simulation of $\mathcal{F}_{\text{PPA}}$.

The inputs of the operator O to both the Outsource- and the Update-task do not contain any secrets, so $\mathcal{S}_{19}$ calls $\mathcal{F}_{\text{PUBA}}$ in the name of O after seeing the first message of an Outsource-task and immediately after the start of the simulation of the Update-task.

H_{20} Instead of relying on the leaked input $\texttt{in}_{\mathcal{TSA}}$ and the received data from the operator to compute whether or not the function parameters *fp* should be accepted during the SignFP task, $\mathcal{S}_{20}$ accepts the function parameters if the functionality returs ok to the operator after both parties sent their inputs. The trusted signing authority is replaced with a genuine dummy party which does not leak $\texttt{in}_{\mathcal{TSA}}$ to the simulator anymore.

H_{21} Instead of relying on the leak of *lin* send by the semi-dummy user U during simulation of the Update-task for honest users, $\mathcal{S}_{21}$ now uses the leaks on the *ssid* provided by $\mathcal{F}_{\text{PUBA}}$ to infer the correct linking number. The honest users are replaced by dummy users, i.e. they only forward their input obtained by $\mathcal{Z}$ to $\mathcal{F}_{\text{PUBA}}$. The remaining interactions with the operator are simulated using the simulator's knowledge. Therefore, the simulator uses its mapping from *ssid* to *lin* to get the correct linking number *lin* during the Update-task.

We now show, by a series of lemmata, that no environment $\mathcal{Z}$ can distinguish the real execution (H_1) from the simulated version in the ideal world (H_{21}) by proving indistinguishability of each pair of consecutive games.

Lemma A.3.2 (Indistinguishability of H_1 and H_2)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_1 and H_2 with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. We are using the UC composition theorems, here. Letting $\mathcal{S}_2$ emulate $\mathcal{F}_{\text{PPA}}$, $\mathcal{F}_{\text{KE}}$ and $\mathcal{F}_{\text{BB}}$ is possible, as all of them are UC-functionalities. As such, they can be replaced by a protocol, for which a simulator $\mathcal{S}_{\mathcal{F}_{\text{PPA}}}$, $\mathcal{S}_{\mathcal{F}_{\text{KE}}}$ and $\mathcal{S}_{\mathcal{F}_{\text{BB}}}$, respectively, exists that provides an indistinguishable view against *any environment* $\mathcal{Z}_{\mathcal{F}_{\text{PPA}}}$ and $\mathcal{Z}_{\mathcal{F}_{\text{BB}}}$. $\mathcal{S}_{\mathcal{F}_{\text{PPA}}}$ is a probabilistic polynomial time (PPT) algorithm that can be executed internally by $\mathcal{S}_2$. This neither breaks the requirement of $\mathcal{S}_2$ being PPT —as every party executing the protocol can only perform polynomially many calls to $\mathcal{F}_{\text{PPA}}$, for each of which $\mathcal{S}_2$ requires polynomial-time to successfully simulate —nor does it reduce the distinguishing advantage of $\mathcal{Z}$.

Assume for the sake of contradiction that there is an environment $\mathcal{Z}$ that can differentiate an honest execution of $\mathcal{F}_{\text{PPA}}$ in H_1 from a simulated execution of $\mathcal{F}_{\text{PPA}}$ by the simulator $\mathcal{S}_2$ in H_2. We can easily use this environment to build one that can differentiate between an execution of $\mathcal{F}_{\text{PPA}}$ in the real world and an ideal execution, where $\mathcal{S}_{\mathcal{F}_{\text{PPA}}}$ provides the view for $\mathcal{Z}_{\mathcal{F}_{\text{PPA}}}$. This works by *encapsulating* parties: Our new distinguishing environment $\mathcal{Z}_{\mathcal{F}_{\text{PPA}}}$ contains the environment $\mathcal{Z}$ that successfully distinguished H_1 and H_2, a set of users $\{U\}$, a proxy P and an operator O, all of which acting according to the protocol specified in H_1. $\mathcal{Z}_{\mathcal{F}_{\text{PPA}}}$ essentially forwards everything that $\mathcal{Z}$ says to the respective parties. After $\mathcal{Z}$ decides on the game, $\mathcal{Z}_{\mathcal{F}_{\text{PPA}}}$ adapts its choice: If $\mathcal{Z}$ outputs H_1, $\mathcal{Z}_{\mathcal{F}_{\text{PPA}}}$ outputs `real` to indicate that this is in the real world. If $\mathcal{Z}$ outputs H_2, $\mathcal{Z}_{\mathcal{F}_{\text{PPA}}}$ outputs `ideal` to indicate that this is an execution in the ideal world.

It is easy to see that the success probability of $\mathcal{Z}_{\mathcal{F}_{\text{PPA}}}$ in deciding whether is in the ideal or real world is equivalent to that of $\mathcal{Z}$ in deciding whether it is playing H_1 or H_2. Hence, if $\mathcal{Z}$ has a non-negligible advantage over guessing, we found an environment that breaks the UC-security assumption of $\mathcal{F}_{\text{PPA}}$.

Note that the same line of argumentation also works for $\mathcal{F}_{\text{KE}}$ and $\mathcal{F}_{\text{BB}}$, which are also assumed to be UC-secure. □

Lemma A.3.3 (Indistinguishability of H_2 and H_3)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_2 and H_3 with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. The behavior of the two simulators is exactly identical, only that $\mathcal{S}_3$ has more information; namely the map $f_{\text{UI}}^{(O)}$. Since none of the messages depend on f_{UI}, indistinguishability trivially follows.

Since there is an abort-criteria for the simulator, we also have show that the abort by $\mathcal{S}_3$ in H_3 only occurs iff P aborts in H_2. Assume that P and O in H_2 have respective shares $\langle\cdot\rangle^P$ and $\langle\cdot\rangle^O$. W.l.o.g., assume that the share that caused the abort of $\mathcal{F}_{\text{PPA}}$ in H_2 be that of the user history $\mathcal{UH}$, as the cases for $\mathtt{in}_U$, o_{OUT}, o_{UNV}, o_α, $\mathsf{o}_\mathbf{s}$ and $\mathsf{o}_\mathbf{a}$ are analogous. In H_2, the two shares are handed over to the subfunctionality $\mathcal{F}_{\text{PPA}}$. There, they are merged with Π_{Share}-*Combine*, which aborts if the verification of the shares fails.

The simulator $\mathcal{S}_3$ in H_3 does exactly the same steps; it aborts, iff verification in Π_{Share}-*Combine* fails. Hence, the abort criteria are identical, the same code is executed by two different machines. So no environment $\mathcal{Z}$ can distinguish the two games. □

Lemma A.3.4 (Indistinguishability of H_3 and H_4)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_3 and H_4 with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Indistinguishability trivially follows from the trapdoor-nature of COM and POK. If any environment $\mathcal{Z}$ could distinguish the execution of the protocol when using *crs* created by $crs \leftarrow \mathsf{SetupPoK}$ from *crs* created by $(crs, td_{\text{sim}}) \leftarrow \mathsf{SetupSPoK}$ with probability $\frac{1}{2} + \mathcal{A}$, we can build a PPT-environment $\mathcal{Z}'$ that breaks the indistinguishability of the dual-mode property of POK, by having $\mathcal{Z}'$ execute the code of all parties in its head. This leads to the

same success probability of $\frac{1}{2} + \mathcal{A}$, thus causing $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$ by requirement of the chosen POK-scheme. □

Lemma A.3.5 (Indistinguishability of H_4 and H_5)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_4 and H_5 with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Any PPT-environment $\mathcal{Z}$ that could distinguish those two games would trivially be able to successfully break the dual-mode property of POK, which is not possible by assumption. □

Lemma A.3.6 (Indistinguishability of H_5 and H_6)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_5 and H_6 with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. As already stated in the game description the only used value that is visible to the PPT environment is the commitment $com_{\mathbf{a}}$. The values $\mathbf{a}$ and $decom_{\mathbf{a}}$ are only used as part of the witness in the zero-knowledge proof in the original protocol and are not used since H_5. Hence, the environment can only see the commitment $com_{\mathbf{a}}$.

For the PPT-environment $\mathcal{Z}$ distinguishing H_5 from H_6 comes down to breaking the hiding-property of the commitment scheme COM which is not possible (for the PPT-environment) by requirement. □

Lemma A.3.7 (Indistinguishability of H_6 and H_7)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_6 and H_7 with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. First, note that since H_5 the zero-knowledge proofs Π are simulated using td_{sim} instead of proving actual properties of the commitments. Hence, the commitments can be exchanged by zero-commitments $\mathsf{COM.commit}(\mathbf{0})$. Other than that, the commitments are only ever used for homomorphic addition, to which the environment $\mathcal{Z}$ only sees committed values. Any PPT-environment $\mathcal{Z}$ that could distinguish the two games H_6 and H_7 would be able to successfully break the hiding property of the commitment scheme COM, which by assumption is only possible with negligible advantage. □

Lemma A.3.8 (Indistinguishability of H_7 and H_8)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_7 and H_8 with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. The proxy sends the same messages in both games. None of the messages that $\mathcal{S}_8$ sends depend in any way on the leak provided by P'. The leak is hidden from the environment. Hence, the distributions for both games are trivially equivalent. □

Lemma A.3.9 (Indistinguishability of H_8 and H_9)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_8 and H_9 with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. In both games, the same messages are sent. The simulator also behaves equivalently, as none of the messages that $\mathcal{S}_9$ sends depend on the information stored in f_{OI}. Hence, no (PPT) environment $\mathcal{Z}$ can differentiate the two games. □

Lemma A.3.10 (Indistinguishability of H_9 and H_{10})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_9 and H_{10} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Again, the only difference is that the simulator obtains and stores additional information. As no messages depend on the additional information, the environment $\mathcal{Z}$ is unable to differentiate the two games. □

Lemma A.3.11 (Indistinguishability of H_{10} and H_{11})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{10} and H_{11} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Through Π_{Share}-*Share*, the user creates one-time pad (OTP) encrypted inputs for P and O; the dummy-adversary $\mathcal{D}$ (hence also $\mathcal{Z}$) only ever sees the share of the corrupted operator, not that of the proxy. $\mathcal{Z}$ only has a partial view, which information-theoretically hides the value that is to be shared due to properties of the one-time pad (OTP). The share itself is not used directly in any further arithmetic computations—only as input to the subfunctionality $\mathcal{F}_{\text{PPA}}$ during the Analytics-task. There, the simulator does not work with the shares, but with the actual values to simulate $\mathcal{F}_{\text{PPA}}$ by computing Δ.
Hence, differentiation between shares of x and shares of $\mathbf{0}$ is not possible for any PPT-environment $\mathcal{Z}$ without breaking the security of one-time pad (OTP) encryption.

The computation of the linking number happens via Blum coin toss, where no secrets are involved, meaning that $\mathcal{S}_{11}$ can easily simulate this part by following the protocol honestly. This change is hence only cosmetical; the same code is executed on a different machine. Hence, since the simulator performs the honest computation and does exactly the same as P' would, the distributions for both games regarding computation of *lin* are equivalent.

Since the shares are created by the simulator, $\mathcal{S}_{11}$ does not have to rely on leaks by P', but can store them directly, having the same information afterwards. □

Lemma A.3.12 (Indistinguishability of H_{11} and H_{12})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{11} and H_{12} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. First, note that honest user leaking information does not change anything in the distribution of sent messages, which makes it impossible for the environment $\mathcal{Z}$ to distinguish. Hence, for indistinguishability, we only have to show that, given the correct linking number *lin*, the simulator for the honest-user case has all the information necessary to report the same messages that U would have sent in H_{11}.

During the Update-tasks with honest user U in game H_{11}, the following tasks are now performed by $\mathcal{S}_{12}$:

- Depending on α and $\mathbf{s}$ (both of which are stored in the clear in f_{UI} which is maintained by the simulator since H_3) the simulator either has to:
  - Send $(\overline{com_{\mathcal{UH}}},\ com'_{\mathcal{UH}},\ com''_{\mathcal{UH}},\ \overline{com_{id}},\ ser,\ lin,\ \Pi,\ \Pi_{Tr},\ com^{(U)}_{ser^{\text{new}}})$ as U to O:
    All the commitments $(\overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \overline{com_{id}}$ and $com^{(U)}_{ser^{\text{new}}})$ are commitments to a zero-vector (due to H_7) and both proofs Π and Π_{Tr} are simulated using td_{sim} (due to H_5). The old serial number ser is independent of the proof and drawn uniformly random by the simulator (since H_7). The new serial number $com^{(U)}_{ser^{\text{new}}}$ is a zero-commitment (since H_7). Finally, note that the linking number lin is leaked by the semi-dummy user. Or, for trivial permutation and direct update vectors:
  - Send $(\overline{com_{\mathcal{UH}}},\ \overline{com_{id}},\ ser,\ lin,\ \Pi,\ com^{(U)}_{ser^{\text{new}}})$ as U to O:
    As this message is a subset of the other message its simulatability automatically follows.
- Verifying outputs:
  As the simulator received the message from O and has the proxy's shares stored in f_{UI}, $\mathcal{S}_{12}$ can follow the honest protocol and abort whenever an honest user would. Hence, we have to show that $\mathcal{S}_{12}$ aborts in H_{12} iff U aborts in H_{11}.

  Since in both H_{11} and H_{12}, the proxy P is assumed to be honest, they send the correct values to the user. Those values are exactly the same as the ones that $\mathcal{S}_{12}$ has stored in f_{UI}. Hence, the simulator already has the user's view on those values. Additionally, the simulator sees messages exchanged between parties; so it also has access to $ct_{\mathtt{out}_U}$. Since the simulator essentially follows the honest protocol from here on, the abort-criteria remain equivalent.

□

Lemma A.3.13 (Indistinguishability of H_{12} and H_{13})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{12} and H_{13} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Since there is no direct link between $\mathcal{Z}$ and $\mathcal{F}_{\text{PUBA}}$, corrupted parties do not get to access $\mathcal{F}_{\text{PUBA}}$. Honest parties only act according to the protocol from H_{12}, which does not contain any interaction with $\mathcal{F}_{\text{PUBA}}$. Hence, there is no way for $\mathcal{Z}$ to distinguish the two games, as every action any honest party takes in H_{12} is equivalent to their actions in H_{13} and the corrupted parties act entirely independent of $\mathcal{F}_{\text{PUBA}}$. □

Lemma A.3.14 (Indistinguishability of H_{13} and H_{14})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{13} and H_{14} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Indistinguishability easily follows from the fact that the TSA $\mathcal{TSA}$ leaks its input to the simulator who performs the same code; hence, the simulator can execute the protocol of $\mathcal{TSA}$ perfectly. Concretely, during the Init-task, the only input to $\mathcal{TSA}$ is `init`, with

the key generation algorithm SIG.SIG.Keygen is public knowledge, and for the SignFP-task, the only inputs are $(\texttt{signfp},\ \texttt{in}_{\mathcal{TSA}})$, and the protocol can be simulated when knowing $\mathsf{sk}_{\mathcal{TSA}}$ (which $\mathcal{S}_{14}$ does from simulation of the Init-task).

Thus, the distribution visible by $\mathcal{Z}$ is identical and the game hop is purely cosmetical. Our claim follows. □

Lemma A.3.15 (Indistinguishability of H_{14} and H_{15})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{14} and H_{15} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Indistinguishability follows from the EUF-CMA security of SIG: Let $\mathcal{Z}$ be an environment that distinguishes between H_{14} and H_{15} with probability $1/2 + \mathcal{A}$ with *non-negligible* advantage $\mathcal{A}$. From $\mathcal{Z}$, we construct an adversary $\mathcal{A}$ on the EUF-CMA property of SIG. Let C be the EUF-CMA challenger. C provides a signature oracle to $\mathcal{A}$. $\mathcal{A}$ flips a random coin and simulates either H_{14} or H_{15}. To that end, $\mathcal{A}$ creates all secrets honestly, except for $\mathsf{sk}_{\mathcal{TSA}}$ and $\mathsf{vk}_{\mathcal{TSA}}$. On every executiton of the SignFP-task, whenever $\mathcal{A}$ is supposed to sign $(\texttt{task},\ com_{fp})$ for O using $\mathsf{sk}_{\mathcal{TSA}}$, $\mathcal{A}$ forwards $(\texttt{task},\ com_{fp})$ to the signature-oracle and uses the result as signature.

Now note that we assume that $\mathcal{Z}$ successfully distinguishes the two games H_{14} and H_{15} notably better than by guessing. Since the only difference is in the way signatures are handled, any distinguishing attack would require $\mathcal{Z}$ to input some distinguishing commitment com_{fp} on function parameters fp into the game, which cause a different simulation.

First, note that if the signature on com_{fp} is accepted in H_{15}, it is also accepted in H_{14}, as $(com_{fp},\ \sigma_{fp}) \in f_{\mathrm{FP}}(k)$ implies that the SignFP task has been called with input $(fp,\ com_{fp})$ successfully and yielded signature σ_{fp}. However, the other way is not as clear; the only differing behavior that can be caused (and used by $\mathcal{Z}$ to detect the change) is by preparing some tuple $(com_{fp},\ \sigma_{fp})$ that is rejected in H_{15} but accepted in H_{14}. Clearly, the latter implies that the signature σ_{fp} on $(\texttt{task},\ com_{fp})$ is valid. The former, however, implies that $\mathcal{Z}$ never called the SignFP-task on fp for function k in the name of O. This means that (1) $\mathcal{A}$ never called the challenge oracle on input $(\texttt{task},\ com_{fp})$, and (2) The signature provided by $\mathcal{Z}$ on $(\texttt{task},\ com_{fp})$ verifies. Taking both together makes this a valid forgery, with which $\mathcal{A}$ can break the EUF-CMA property of SIG. □

Lemma A.3.16 (Indistinguishability of H_{15} and H_{16})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{15} and H_{16} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. The change induced in this game only *activates* the functionality $\mathcal{F}_{\mathrm{PUBA}}$. Since, at this point, it is not accessed by any honest party and neither the environment, nor the dummy adversary can access $\mathcal{F}_{\mathrm{PUBA}}$, indistinguishability between the two games trivially follows. □

Lemma A.3.17 (Indistinguishability of H_{16} and H_{17})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{16} and H_{17} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. First, note that the environment $\mathcal{Z}$ is unable to differentiate between the case where the honest user (i.e. the ones that are not controlled by $\mathcal{Z}$) forwards their input to $\mathcal{F}_{\text{PUBA}}$ and the one where they do not, as there is no direct line of communication between $\mathcal{F}_{\text{PUBA}}$ and either the corrupted parties, or the environment. Hence, this change is impossible to detect for any environment $\mathcal{Z}$.

We claim indistinguishability of the remaining changes based on the fact that the simulator $\mathcal{S}_{17}$ sends exactly the same messages that an honest user would in H_{16}. Using the leaks, the simulator in H_{17} can create every message that the user would have sent in H_{16}, as we will show now:

UReg. Here, the user has no secret input; the simulator can safely draw a random key similar to the real user and respond to calls to $\mathcal{F}_{\text{BB}}$ with pk_U. The verification step is only based on the messages, which the simulator sees; this means, that $\mathcal{S}_{17}$ can abort in H_{17} whenever U aborts in H_{16}.

Bookkeeping. The first message that $\mathcal{S}_{17}$ has to send here in the name of U is $(\overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \Pi_{Val}, com^{(U)}_{ser^{\text{new}}})$.

Since H_7, the commitments are commitments on the all-zero vector and hence independent of the users secret inputs and can be simulated by $\mathcal{S}_{17}$. Since H_5, $\mathcal{S}_{17}$ simulates the zero-knowledge proof Π_{Val}, which is possible without knowing the witness. The old serial number, ser, is independent of both $com_{\mathcal{UH}}$ and Π_{Val}, and can be drawn uniformly random.

In case any non-trivial permutation α or set vector $\mathbf{s}$ is returned from the simulation of $\mathcal{F}_{\text{PPA}}$ and hence the computation of the application-specific function Δ, $\mathcal{S}_{17}$ has to report a second message in which it proves correct update of the user history. This results in the message $(com'_{\mathcal{UH}}, com''_{\mathcal{UH}}, \Pi_{Tr})$. The commitments are once again all-zero commitments due to H_7. The proof Π_{Tr} is simulated due to H_5 but requires knowledge of the full statement. This statement contains $\overline{com_{\mathcal{UH}}}, com'_{\mathcal{UH}}, com''_{\mathcal{UH}}$ (which were chosen as commitments on the all-zero vector due to H_7), and α and $\mathbf{s}$ (which are also required in order to decide if the second message has to be reported at all; note that both α and $\mathbf{s}$ are known by O and hence cannot be forged or set arbitrarily). The latter is obtained by simulation of $\mathcal{F}_{\text{PPA}}$ and hence consistent with the operators view. Hence $\mathcal{S}_{17}$ can reconstruct the statement of the second proof according to $\mathcal{R}_{Tr}$ which proves that the output of $\mathcal{F}_{\text{PPA}}$ has been transferred to the user history accordingly. Again, we stress that this message is only reported for non-trivial values of α and $\mathbf{s}$.

Finally, the simulator has to perform the verification step, as $\mathcal{S}_{17}$ always used zero-vectors for shares and can hence verify the values received from O. This is done by executing the honest protocol, namely by calling Π_{Verify} with an honestly created logbook Log.

Outsource. During an Outsource-task, the shares have been created since H_{11} by the simulator anyways. Since H_5 and H_7 the simulator also creates the commitments and the zero-knowledge proof. The serial is again drawn at random since H_7 as

it is independent of $com_{ser}(= com_0)$. Hence, the first message, $(\langle \mathcal{UH} \rangle^O, \langle \mathtt{in}_U \rangle^O, \langle \mathtt{o}_{\mathrm{OUT}} \rangle^O, \langle \mathtt{o}_{\mathrm{UNV}} \rangle^O, \langle \mathtt{o}_\alpha \rangle^O, \langle \mathtt{o}_\mathbf{s} \rangle^O, \langle \mathtt{o}_\mathbf{a} \rangle^O, com^{(O)}_{\mathcal{UH}}, ser, \overline{com_{id}}, \Pi, com^{(U)}_{ser^{\mathrm{new}}})$ is indistinguishable from the one in H_{16}.

Verification, again, is only dependent on what the simulator knows already and hence can be simulated by executing the honest protocol.

Update. The Update-task has been simulated already since H_{12}.

For security against a corrupted user U our definition of $\mathcal{F}_{\mathrm{PUBA}}$ requires $\mathcal{S}_{17}$ to call $\mathcal{F}_{\mathrm{PUBA}}$ in the name of U only during the Outsource-task and not during any of the other tasks:

- In **UReg**, all that would change is that $\mathcal{F}_{\mathrm{PUBA}}$ adds U to the list of known users, which is never checked against corrupted users. Interaction with $\mathcal{F}_{\mathrm{BB}}$ might still take place, though, but that one is simulated since H_2.
- In the **Bookkeeping**-task the environment essentially talks to itself. Updates to $\mathcal{UH}$ can be done by $\mathcal{Z}$ without access to $\mathcal{F}_{\mathrm{PUBA}}$ as the operator can sign any user history $\mathcal{UH}$ and thus create a valid new logbook Log. Simulation of $\mathcal{F}_{\mathrm{PPA}}$ occurs outside of the actual protocol since $\mathcal{S}_{17}$ plays $\mathcal{S}_{\mathcal{F}_{\mathrm{PPA}}}$ since H_2.
- For **Outsource** $\mathcal{F}_{\mathrm{PUBA}}$ stores the corrupted user's information in a list, alongside that of honest users, and which are used later for the Analytics-task. There, it does not make a difference if the corrupted user's $\mathcal{UH}$ is input to $\mathcal{F}_{\mathrm{PUBA}}$ directly, or if it is equivocated during simulation of the subsequent Analytics task.
- During **Analytics** the simulator receives input from O to $\mathcal{F}_{\mathrm{PPA}}$ which contains the shares the corrupted user U prepared for the operator: $(\mathtt{analyt}k, \{(\langle \mathcal{UH} \rangle^O, \langle \mathtt{in}_U \rangle^O, \langle \mathtt{o}_{\mathrm{OUT}} \rangle^O, \langle \mathtt{o}_{\mathrm{UNV}} \rangle^O, \langle \mathtt{o}_\alpha \rangle^O, \langle \mathtt{o}_\mathbf{s} \rangle^O, \langle \mathtt{o}_\mathbf{a} \rangle^O)_z\}_{z=1}^{Z_k}, fp, decom_{fp}, \mathtt{in}_O)$. The respective shares of the proxy were already stored in $f_{\mathrm{OI}}(pid_P, k)$ during simulation of the Outsource-task. Hence all shares can be reconstructed by $\mathcal{S}_{17}$ using the protocol Π_{Share}-*Combine*. This value is equivalent to the input corrupted users that would have input to $\mathcal{F}_{\mathrm{PUBA}}$. So $\mathcal{S}_{17}$ can provide $\mathcal{F}_{\mathrm{PUBA}}$ with the correct tuples $(\mathcal{UH}, \mathtt{in})$ for corrupted users.

  Correctness trivially follows as the simulator in H_{17} has, at this point, exactly the same values that are divided to P and O in H_{16}. Hence, any attempt by $\mathcal{Z}$ to manipulate data that would have worked in H_{16} also works in H_{17} and vice versa, making it impossible for any PPT-environment $\mathcal{Z}$ to distinguish.
- During the **Update**-task corrupted users only obtain masked values from the proxy. This step does not have any consequences for further interaction as $\mathcal{F}_{\mathrm{PUBA}}$ only loads and returns data that is not accessed at any further point throughout the lifetime of the system. Hence, it suffices to call $\mathcal{F}_{\mathrm{PUBA}}$ in the name of *any* corrupted user to have it deliver output to the Proxy.

□

Lemma A.3.18 (Indistinguishability of H_{17} and H_{18})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{17} and H_{18} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Intuitively, the proxy does not have any secret inputs. It only obtains information via secret shares of the user. At this point, the simulator $\mathcal{S}_{18}$ already simulates all the messages sent by the user and hence can act as the honest proxy would by following the protocol from H_{17}.

In more detail, the situation for an *honest* user is as follows:

Outsource. Since the proxy is now played by the simulator who also sends all messages on behalf of the user, all messages from user to proxy and vice versa can be ignored. This automatically resolves most of the Outsource-task; the only interaction between the proxy and the operator that has to be simulated is during a Blum coin toss to create *lin*, which does not depend on any secret inputs at all. This can honestly be executed by $\mathcal{S}_{18}$.

Managing f_{OI} is also trivially possible, as all the values stored there come from the user, who is played by the simulator at this point, anyway. Correctness of the values follows from the fact that $\mathcal{S}_{18}$ only stores information there in H_{18} when the honest proxy P in H_{17} would.

Analytics. In H_{17}, the proxy loads the first Z_k entries of f_{OI} before calling $\mathcal{F}_{\mathrm{PPA}}$. We already argued for the Outsource-task that the information $\mathcal{S}_{18}$ stores in f_{OI} in H_{18} is equivalent to what P stores during H_{17}; hence, $\mathcal{S}_{18}$ can simulate by following the honest protocol.

Storing the information in $f_{\mathrm{UI}}^{(P)}$ after the simulation of $\mathcal{F}_{\mathrm{PPA}}$ is not required anymore, as it only contains information that the simulator can infer from f_{UI}.

Update. The protocol for the Update-task consists of two disjoint parts: the interaction between the user and the proxy and the interaction between the user and the operator.

The latter is independent of anything the proxy does. The former is independent of anything the environment $\mathcal{Z}$ (or the dummy-adversary $\mathcal{D}$) do and does not have to be simulated, as $\mathcal{Z}$ does not have the ability to read messages exchanged between honest parties.

For a corrupted users the changes for the Outsource- and Update-tasks are as follows:

Outsource. Here, everything works by following the protocol of P honestly. The simulator receives the shares, which are send from U to P. Those can be stored in $f_{\mathrm{OI}}(pid_P, k)$.

Since the simulator now also performs the Blum coin toss, $\mathcal{S}_{18}$ knows *lin* and can send it to the user directly.

Update. The only interaction that has to be simulated is the part where P, after receiving lin from U, sends $(ct_\alpha,\ ct_{\mathbf{s}},\ ct_{\mathbf{a}},\ ct_{decom_{\mathbf{a}}},\ ct_{\mathrm{out}_U})$ to U. During simulation of the task for Analytics both values were created honestly and stored in $f_{\mathrm{UI}}(lin)$. Since lin is obtained from U, the proxy in H_{17} will send exactly the same message as the simulator in H_{18}.

This shows that the two games are indistinguishable for all PPT-environments $\mathcal{Z}$. □

Lemma A.3.19 (Indistinguishability of H_{18} and H_{19})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{18} and H_{19} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Indistinguishability of the first change, namely that computations of Δ are replaced by leaks from $\mathcal{F}_{\mathrm{PUBA}}$, follows from the following facts:

1. In both hybrid games, the result of the computation is based on the output of Δ. The simulator performs this computation in H_{18}; there, it acts honestly according to the simulator description, hence $\mathcal{S}_{19}$ would not cheat. The functionality $\mathcal{F}_{\mathrm{PUBA}}$ is, by UC-conventions, modeled as an incorruptible entity and hence performs the same honest computation. Hence, both parties compute Δ honestly.
2. Both parties, $\mathcal{F}_{\mathrm{PUBA}}$ in H_{19} and the simulator in H_{18}, use exactly the same input. Until the point where the interaction takes place, the simulators in both H_{18} and H_{19} behave equivalently, leading to an identical view for $\mathcal{Z}$. From there on, the simulator in H_{18} performs the computation $\Delta(\cdot)$ directly and obtains the outputs. In H_{19}, $\mathcal{S}_{19}$ forwards the same input to $\mathcal{F}_{\mathrm{PUBA}}$ (which is possible because both have the same interface). The functionality then uses this input in order to compute Δ. The outputs are leaked to $\mathcal{S}_{19}$ via functionality output, who now has exactly the same values as in H_{18} and can continue equivalently.

Hence, the parts involving interaction with $\mathcal{F}_{\mathrm{PPA}}$ remain equivalent, as essentially the output data that $\mathcal{S}_{19}$ obtains in H_{19} via leakage has been created similarly to the output that the simulator computed in H_{18}.

We now have to show that the abort-criteria remain equivalent. In H_{19}, $\mathcal{S}_{19}$ aborts if the input of O to $\mathcal{F}_{\mathrm{PPA}}$ differs from the shares that U sent to O during the Outsource-task. In H_{18}, the user aborts via $\mathcal{F}_{\mathrm{PPA}}$, as shares that were changed by the operator would be recognized as forgery due to Π_{Verify} with overwhelming probability.

Next, we have to prove indistinguishability of the equivocation step for corrupted users. To that end, we claim that the updated input shares that $\mathcal{S}_{19}$ inputs to $\mathcal{F}_{\mathrm{PUBA}}$ are the same that were shared during the Outsource-task and that would have been used by the proxy and operator in a real execution. The proxy-shares were treated similar to a real execution, as the simulator stored the message that was received during the Outsource-task in f_{OI}. The operator shares were taken from Os input to $\mathcal{F}_{\mathrm{PPA}}$ and are hence also visible to the simulator. Hence, the same values that would have been taken as input for Δ in H_{18} are also taken by $\mathcal{F}_{\mathrm{PUBA}}$ in H_{19}. As mentioned above, $\mathcal{F}_{\mathrm{PUBA}}$ computes Δ just as was done in previous games, so no environment can distinguish. □

Lemma A.3.20 (Indistinguishability of H_{19} and H_{20})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{19} and H_{20} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Again this gamehop is only cosmetical as the same code still is executed on the same inputs, but from a different machine. In H_{19} the simulator obtains $\mathtt{in}_{\mathcal{TSA}}$ from the (semi-)dummy TSA and (fp, com_{fp}, $decom_{fp}$, $\mathtt{task}$, $\mathtt{in}_O$) from the corrupted operator and then performs some consistency checks regarding the FPs before evaluating Δ. In H_{20} the simulator only obtains (fp, com_{fp}, $decom_{fp}$, $\mathtt{task}$, $\mathtt{in}_O$) from the corrupted operator and then performs the same consistency checks regarding the FPs before inputting (fp, $\mathtt{task}$, $\mathtt{in}_O$) to $\mathcal{F}_{\text{PUBA}}$. The input from the TSA—$\mathtt{in}_{\mathcal{TSA}}$—was already forwarded by the honest dummy party to $\mathcal{F}_{\text{PUBA}}$ so the inputs are the same.

Indistinguishability thus follows directly. □

Lemma A.3.21 (Indistinguishability of H_{20} and H_{21})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{20} and H_{21} with probability $1/2 + \mathcal{A}$. It holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. The situation only changes during Update-tasks with an honest user, where the simulator $\mathcal{S}_{21}$ in the honest-user setting does not receive the leak lin from U, but instead the subsession id $ssid$ of the respective Outsource-instance from $\mathcal{F}_{\text{PUBA}}$. Since the simulator updated the list f_{LN} correctly during the simulation of the Outsource-task, $\mathcal{S}_{21}$ can obtain the same linking number lin in H_{21} that U has sent in H_{20}.

Given that $\mathcal{F}_{\text{PUBA}}$ is, by definition, incorruptible, it will always send the correct subsession id $ssid$ to the simulator $\mathcal{S}_{21}$. During the Outsource-task, the linking number lin was honestly created by the simulator $\mathcal{S}_{21}$ in an interaction with O. This number is used by all parties as linking number and is stored in f_{LN} just as honest parties would store it. During the Update-task, an honest user would look up their linking number, which can be simulated by following the program of U and fetching it from f_{LN}.

Hence, there is no way that the linking number an honest user would store during an Outsource- and later reveal during an Update-task in H_{20} would differ from the linking number that the simulator stores (and later reveals) during the simulation of H_{21}. □

The final game, H_{21}, corresponds to our ideal world. Since we have shown that no efficient environment $\mathcal{Z}$ can differentiate this from the real execution in H_1, our corollary follows:

Corollary A.3.22 (User Security)**.** *For all environments $\mathcal{Z}$ who statically corrupted the operator, it follows that*

$$\Pi_{\text{PUBA}}{}^{(\mathcal{F}_{\text{PPA}}, \mathcal{F}_{\text{BB}}, \mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{SMT}}^{\text{os-auth}}, \mathcal{F}_{\text{CRS}})} \geq_{UC} \mathcal{F}_{\text{PUBA}}$$

We have shown in Lemma A.3.2 to Lemma A.3.21, that under static corruption of the operator, the simulator $\mathcal{S}_{\text{USec}}$ acting in the ideal world can provide a view for $\mathcal{Z}$ that is indistinguishable from a real execution of the protocol:

$$view_{\mathcal{Z},\mathcal{A},\Pi_{\text{PUBA}}} \approx^{c} view_{\mathcal{Z},\mathcal{S}_{\text{USec}},\mathcal{F}_{\text{PUBA}}}$$

A.3.2. System Security

This section contains an investigation of the remaining corruption scenarios, namely the ones that are relevant to maintaining privacy of an honest *operator*. That is, we consider scenarios where any subset of users and proxies can be corrupted and present a simulator, which provides a view in the ideal world that cannot be distinguished from a real-world execution.

Simulator $\mathcal{S}_{\text{SysSec}}$ for corrupted users and proxies

Π-*Shared State*:
- Trapdoor td_{ext} for extracting proofs.
- Signature key pair $(\mathsf{vk}_O, \mathsf{sk}_O)$
- Signature key pair $(\mathsf{vk}_{\mathcal{TSA}}, \mathsf{sk}_{\mathcal{TSA}})$
- List $\mathsf{L}_{\mathsf{SER}}$ of observed serial numbers.
- Partial mapping f_{ID} on $\mathbb{G}_1$: $\mathsf{pk}_U \mapsto pid_U$ that maps user public keys pk_U to the pid of the corresponding user
- Mapping $f_{\text{OI}}^{(P)}$ on $\{1, \ldots, K\}$ that maps k to a list $f_{\text{OI}}^{(P)}(k)$ of entries $(lin, \langle \mathcal{UH} \rangle^P, \langle \mathtt{in}_U \rangle^P, \langle \mathsf{o}_{\text{OUT}} \rangle^P, \langle \mathsf{o}_{\text{UNV}} \rangle^P, \langle \mathsf{o}_\alpha \rangle^P, \langle \mathsf{o}_\mathbf{s} \rangle^P, \langle \mathsf{o}_\mathbf{a} \rangle^P)$ for every honest proxy P
- Mapping $f_{\text{OI}}^{(O)}$ on $\{pid_P\} \times \{1, \ldots, K\}$ that maps (pid_P, k) to a list $f_{\text{OI}}^{(O)}(pid_P, k)$ of entries $(lin, \langle \mathcal{UH} \rangle^O, \langle \mathtt{in}_U \rangle^O, \langle \mathsf{o}_{\text{OUT}} \rangle^O, \langle \mathsf{o}_{\text{UNV}} \rangle^O, \langle \mathsf{o}_\alpha \rangle^O, \langle \mathsf{o}_\mathbf{s} \rangle^O, \langle \mathsf{o}_\mathbf{a} \rangle^O)$
- One partial mapping $f_{\text{UI}}^{(P)}$ on $\{lin\}$ with $lin \mapsto (ct_\alpha, ct_\mathbf{s}, ct_\mathbf{a}, ct_{decom_\mathbf{a}}, ct_{\mathtt{out}_U})$ for every honest proxy P
- Partial mapping $f_{\text{UI}}^{(O)}$ on $\{lin\}$. $lin \mapsto (\alpha, \mathbf{s}, com_\mathbf{a}, ct_{\mathtt{out}_U})$
- Partial mapping f_{LN} on $\{ssid\}$: $ssid \mapsto lin$
- Mapping f_{FP} on $(\{\mathtt{ureg} \cup \mathtt{bookkeep}K \cup \mathtt{analyt}K\} \times L)$ that maps a task $\mathtt{task}$ and a FP-index ℓ to a tuple $(com_{fp}, decom_{fp}, \sigma_{fp})$ of a commitment and a signature.

Π_{PUBA}-*Setup*:
1. Run a modified version of $crs \leftarrow \Pi_{\text{PUBA}}\text{-}Setup(1^\lambda)$
   - $crs_{\text{ZK}} \leftarrow$ SetupPoK is replaced by $(crs_{\text{ZK}}, td_{\text{ext}}) \leftarrow$ SetupEPoK for extracting proofs.

Π_{PUBA}-*Init*:
1. Upon receiving $(\mathtt{init}, pid_O)$ from $\mathcal{F}_{\text{PUBA}}$:
   - Create and store a signature-key pair $(\mathsf{sk}_O, \mathsf{vk}_O) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
   - From now on, reply to $\mathcal{F}_{\text{BB}}$ calls of the form $(\mathtt{Retrieve}, pid_O)$ with vk_O

   $\hookrightarrow$ Allow $\mathcal{F}_{\text{PUBA}}$ to continue.
2. Upon receiving $(\mathtt{init}, pid_{\mathcal{TSA}})$ from $\mathcal{F}_{\text{PUBA}}$:
   - Create and store a signature-key pair $(\mathsf{sk}_{\mathcal{TSA}}, \mathsf{vk}_{\mathcal{TSA}}) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
   - From now on, reply to $\mathcal{F}_{\text{BB}}$ calls of the form $(\mathtt{Retrieve}, pid_{\mathcal{TSA}})$ with $\mathsf{vk}_{\mathcal{TSA}}$

   $\hookrightarrow$ Allow $\mathcal{F}_{\text{PUBA}}$ to continue.

Π_{PUBA}-*SignFP*:

. . . . . .

O honest, $\mathcal{TSA}$ honest:
1. Upon receiving leak $(\mathtt{task}, \ell)$ from $\mathcal{F}_{\text{PUBA}}$:
   - Compute $(com_\mathbf{0}, decom_\mathbf{0}) \leftarrow \mathsf{COM.commit}(\mathbf{0})$.
   - Compute $\sigma_{fp} \leftarrow \mathsf{SIG.Sign}((\mathtt{task}, com_\mathbf{0}), \mathsf{sk}_{\mathcal{TSA}})$.
   - Store $f_{\text{FP}}(\mathtt{task}, \ell) \coloneqq (com_\mathbf{0}, decom_{fp}, \sigma_{fp})$.

Π_{PUBA}-*UReg*:

U and O honest:
1. Draw random user public key $\mathsf{pk}_U \xleftarrow{r} \mathbb{G}_1$
2. From now on, reply to $\mathcal{F}_{\text{BB}}$ calls of the form $(\texttt{Retrieve},\ pid_U)$ with pk_U
3. Report encrypted messages.

U corrupted, O honest:
1. Upon receiving instructions from $\mathcal{Z}$ to send $(\texttt{Retrieve},\ pid_O)$ from U to $\mathcal{F}_{\text{BB}}$:
   ↪ Report output (vk_O) from $\mathcal{F}_{\text{BB}}$ to U.
2. Upon receiving instructions from $\mathcal{Z}$ to send $(\texttt{Register},\ pid_U,\ \mathsf{pk}_U)$ from U to $\mathcal{F}_{\text{BB}}$:
   - If a previous call $(\texttt{Register}, pid_U, \mathsf{pk}'_U)$ to $\mathcal{F}_{\text{BB}}$ for some pk'_U has been previously recorded, abort in the name of O.
3. Upon receiving leak ℓ from $\mathcal{F}_{\text{PUBA}}$:
   - Set[a] $(com_{\mathbf{0}},\ decom_{\mathbf{0}},\ \sigma_{fp}) \coloneqq f_{\text{FP}}(\texttt{ureg},\ \ell)$.
4. Upon receiving instructions from $\mathcal{Z}$ to send $(\Pi, com_{id}, com^{(U)}_{ser^{\text{new}}})$ from U to O:
   - If $f_{\text{ID}}(\mathsf{pk}_U) \neq \perp$ abort, else set $f_{\text{ID}}(\mathsf{pk}_U) = pid_U$
   - $stmt \coloneqq (\mathsf{pk}_U, com_{id})$
   - Check[a] $\mathsf{POK.Verify}(\Pi, stmt, \mathcal{R}_{\mathsf{UR}}) \stackrel{?}{=} 1$

   ↪ Report message $(com_{\mathbf{0}},\ \sigma_{fp})$ from O to U.
5. Upon receiving instructions from $\mathcal{Z}$ to send $(\texttt{ureg}, com_{fp}, \mathtt{in}_U)$ from U to $\mathcal{F}_{\text{PPA}}$:
   - Check[a] $\mathsf{COM.open}(com_{fp},\ decom_{\mathbf{0}},\ \mathbf{0}) = 1$.

   ↪ Call $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{ureg}, \mathtt{in}_U)$ in the name of U.
6. Upon receiving output $(\mathcal{UH}, \mathtt{out}_U)$ from $\mathcal{F}_{\text{PUBA}}$:
   - $(com_{\mathcal{UH}},\ decom_{\mathcal{UH}}) \leftarrow \mathsf{COM.commit}(\mathcal{UH})$
   - $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
   - $(com^{(O)}_{ser^{\text{new}}},\ decom^{(O)}_{ser^{\text{new}}}) \leftarrow \mathsf{COM.commit}((ser^{\text{new}})^{(O)})$
   - $com_{ser^{\text{new}}} \coloneqq com^{(O)}_{ser^{\text{new}}} \oplus com^{(U)}_{ser^{\text{new}}}$
   - $(com_{lin},\ decom_{lin}) \leftarrow \mathsf{COM.commit}(0)$
   - $\sigma \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O,\ com_{\mathcal{UH}} \| com_{ser^{\text{new}}} \| com_{lin} \| com_{id})$

   ↪ Report output $(\mathcal{UH}, decom_{\mathcal{UH}}, \mathtt{out}_U)$ from $\mathcal{F}_{\text{PPA}}$ to U.

   ↪ Report message $(com_{\mathcal{UH}},\ (ser^{\text{new}})^{(O)},\ com^{(O)}_{ser^{\text{new}}},\ decom^{(O)}_{ser^{\text{new}}},\ com_{lin},\ decom_{lin},\ com_{id},\ decom_{id},\ \sigma)$ from O to U.

   ↪ Allow $\mathcal{F}_{\text{PUBA}}$ to continue.

$\{\Pi_{\text{PUBA}}\text{-}Bookkeeping^{(k)}\}_{k=1}^{K}$:

U and O honest: Report encrypted messages.

U corrupted, O honest:
1. Upon receiving leak (ℓ) from $\mathcal{F}_{\text{PUBA}}$ and instructions from $\mathcal{Z}$ to send $(\overline{com_{\mathcal{UH}}},\ ser,\ \overline{com_{lin}},\ \overline{com_{id}},\ \Pi_{Val},\ com^{(U)}_{ser^{\text{new}}})$ from U to O:
   - $stmt_{Val} \coloneqq (\overline{com_{\mathcal{UH}}}, ser, \overline{com_{lin}}, \overline{com_{id}}, \mathsf{vk}_O)$
   - Check[a] $\mathsf{POK.Verify}(crs_{\text{ZK}},\ \mathcal{R}_{Val},\ stmt_{Val},\ \Pi_{Val}) \stackrel{?}{=} 1$
   - Check[a] $ser \notin \mathsf{L}_{\text{SER}}$
   - $\mathsf{L}_{\text{SER}} \coloneqq \mathsf{L}_{\text{SER}} \cup \{ser\}$
   - Set[a] $(com_{\mathbf{0}},\ decom_{\mathbf{0}},\ \sigma_{fp}) \coloneqq f_{\text{FP}}(\texttt{bookkeep}k,\ \ell)$.

   ↪ Report message $(com_{\mathbf{0}},\ \sigma_{fp})$ from O to U.
2. Upon receiving $(\texttt{bookkeep}k, \mathcal{UH}, \overline{decom_{\mathcal{UH}}}, com_{fp}, \mathtt{in}_U)$ from U to $\mathcal{F}_{\text{PPA}}$:
   - Check[a] $\mathsf{COM.open}(com_{fp},\ decom_{\mathbf{0}},\ \mathbf{0}) = 1$.
   - Check[a] $\mathsf{COM.open}(\overline{com_{\mathcal{UH}}}, \overline{decom_{\mathcal{UH}}}, \mathcal{UH}) \stackrel{?}{=} 1$
   - Extract pk_U using $\mathsf{POK.ExtractWit}(td_{\text{ext}},\ \Pi_{Val},\ stmt_{Val},\ \mathcal{R}_{Val})$
   - Load[a] $pid_{U'} \coloneqq f_{\text{ID}}(\mathsf{pk}_U)$
   - Choose U' as the user corresponding to $pid_{U'}$

↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{bookkeep}k,\ \texttt{in}_U)$ in the name of U'

3. Upon receiving output $(\alpha,\ \mathbf{s},\ \mathbf{a},\ \texttt{out}_U)$ from $\mathcal{F}_{\text{PUBA}}$ to U:
   - Compute $(com_{\mathbf{a}},\ decom_{\mathbf{a}}) \leftarrow \mathsf{COM.commit}(\mathbf{a})$

   ↪ Report message $(\alpha,\ \mathbf{s},\ \mathbf{a},\ com_{\mathbf{a}},\ decom_{\mathbf{a}},\ \texttt{out}_U)$ from $\mathcal{F}_{\text{PPA}}$ to U
4. **if** $\alpha \neq \bot \vee \mathbf{s} \neq \bot$ **then**
   - Upon receiving instructions from $\mathcal{Z}$ to send $(com'_{\mathcal{UH}},\ com''_{\mathcal{UH}},\ \Pi_{Tr})$ from U to O:
     - $stmt_{Tr} \coloneqq (\overline{com_{\mathcal{UH}}},\ com'_{\mathcal{UH}},\ com''_{\mathcal{UH}},\ \alpha,\ \mathbf{s})$
     - Check[a] $\mathsf{POK.Verify}(\Pi_{Tr},\ stmt_{Tr},\ \mathcal{R}_{Tr}) \stackrel{?}{=} 1$

   fi
5. Compute new user history.
   - $com^{\text{new}}_{\mathcal{UH}} \coloneqq com''_{\mathcal{UH}} \oplus com_{\mathbf{a}}$
   - $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
   - $(com^{(O)}_{ser^{\text{new}}},\ decom^{(O)}_{ser^{\text{new}}}) \leftarrow \mathsf{COM.commit}((ser^{\text{new}})^{(O)})$
   - $com^{\text{new}}_{ser} \coloneqq com^{(O)}_{ser^{\text{new}}} \oplus com^{(U)}_{ser^{\text{new}}}$
   - $\sigma^{\text{new}} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O,\ com^{\text{new}}_{\mathcal{UH}} \| com^{\text{new}}_{ser} \| \overline{com_{lin}} \| \overline{com_{id}})$

   ↪ Report message $(com^{\text{new}}_{\mathcal{UH}},\ (ser^{\text{new}})^{(O)},\ com^{(O)}_{ser^{\text{new}}},\ decom^{(O)}_{ser^{\text{new}}},\ \sigma^{\text{new}})$ from O to U

   ↪ Allow $\mathcal{F}_{\text{PUBA}}$ to deliver output to O

$\{\Pi_{\text{PUBA}}\text{-}Outsource^{(k)}\}^{K}_{k=1}$:

U, P and O honest:

1. Upon receiving $(\texttt{outsource}k, \text{User})$ from $\mathcal{F}_{\text{PUBA}}$, $(\texttt{outsource}k, pid_P)$ from $\mathcal{F}_{\text{PUBA}}$ and $(\texttt{outsource}k, pid_O)$ from $\mathcal{F}_{\text{PUBA}}$:
   - Append an empty entry $(\bot,\ \ldots,\ \bot)$ to $f^{(P)}_{\text{OI}}(k)$
   - Append an empty entry $(\bot,\ \ldots,\ \bot)$ to $f^{(O)}_{\text{OI}}(pid_P, k)$

   ↪ Allow $\mathcal{F}_{\text{PUBA}}$ to continue.

U corrupted, P and O honest:

1. Upon receiving $(\texttt{outsource}k,\ pid_O)$ from $\mathcal{F}_{\text{PUBA}}$ and $(\texttt{outsource}k,\ pid_P)$ from $\mathcal{F}_{\text{PUBA}}$ and instructions from $\mathcal{Z}$ to send $(\langle\mathcal{UH}\rangle^O,\ \langle\texttt{in}_U\rangle^O,\ \langle\texttt{o}_{\text{OUT}}\rangle^O,\ \langle\texttt{o}_{\text{UNV}}\rangle^O,\ \langle\texttt{o}_\alpha\rangle^O,\ \langle\texttt{o}_{\mathbf{s}}\rangle^O,\ \langle\texttt{o}_{\mathbf{a}}\rangle^O,\ ser,\ \overline{com_{id}},\ \Pi,\ com^{(U)}_{ser^{\text{new}}})$ from U to O and $(\langle\mathcal{UH}\rangle^P,\ \langle\texttt{in}_U\rangle^P,\ \langle\texttt{o}_{\text{OUT}}\rangle^P,\ \langle\texttt{o}_{\text{UNV}}\rangle^P,\ \langle\texttt{o}_\alpha\rangle^P,\ \langle\texttt{o}_{\mathbf{s}}\rangle^P,\ \langle\texttt{o}_{\mathbf{a}}\rangle^P)$ from U to P:
   - $stmt \coloneqq (\langle\mathcal{UH}\rangle^O, com^{(O)}_{\mathcal{UH}}, ser, \overline{com_{id}}, \mathsf{vk}_O)$
   - Check[a] $\mathsf{POK.Verify}(crs_{\text{ZK}}, \mathcal{R}_{\texttt{0S}}, stmt, \Pi) \stackrel{?}{=} 1$
   - Check[a] $ser \notin \mathsf{L}_{\text{SER}}$
   - $\mathsf{L}_{\text{SER}} \coloneqq \mathsf{L}_{\text{SER}} \cup \{ser\}$
   - Combine[b] $\mathcal{UH} \coloneqq \Pi_{\text{Share}}\text{-}Combine(\langle\mathcal{UH}\rangle^O, \langle\mathcal{UH}\rangle^P)$
   - Combine[b] $\texttt{in}_U \coloneqq \Pi_{\text{Share}}\text{-}Combine(\langle\texttt{in}_U\rangle^O, \langle\texttt{in}_U\rangle^P)$
   - Combine[b] $\texttt{o}_{\text{OUT}} \coloneqq \Pi_{\text{Share}}\text{-}Combine(\langle\texttt{o}_{\text{OUT}}\rangle^O, \langle\texttt{o}_{\text{OUT}}\rangle^P)$
   - Combine[b] $\texttt{o}_{\text{UNV}} \coloneqq \Pi_{\text{Share}}\text{-}Combine(\langle\texttt{o}_{\text{UNV}}\rangle^O, \langle\texttt{o}_{\text{UNV}}\rangle^P)$
   - Combine[b] $\texttt{o}_\alpha \coloneqq \Pi_{\text{Share}}\text{-}Combine(\langle\texttt{o}_\alpha\rangle^O, \langle\texttt{o}_\alpha\rangle^P)$
   - Combine[b] $\texttt{o}_{\mathbf{s}} \coloneqq \Pi_{\text{Share}}\text{-}Combine(\langle\texttt{o}_{\mathbf{s}}\rangle^O, \langle\texttt{o}_{\mathbf{s}}\rangle^P)$
   - Combine[b] $\texttt{o}_{\mathbf{a}} \coloneqq \Pi_{\text{Share}}\text{-}Combine(\langle\texttt{o}_{\mathbf{a}}\rangle^O, \langle\texttt{o}_{\mathbf{a}}\rangle^P)$
   - $lin^{\text{new}} \xleftarrow{r} \mathbb{Z}_p$
   - Append $(lin^{\text{new}},\ \langle\mathcal{UH}\rangle^P,\ \langle\texttt{in}_U\rangle^P,\ \langle\texttt{o}_{\text{OUT}}\rangle^P,\ \langle\texttt{o}_{\text{UNV}}\rangle^P,\ \langle\texttt{o}_\alpha\rangle^P,\ \langle\texttt{o}_{\mathbf{s}}\rangle^P,\ \langle\texttt{o}_{\mathbf{a}}\rangle^P)$ to $f^{(P)}_{\text{OI}}(k)$
   - Append $(lin^{\text{new}},\ \langle\mathcal{UH}\rangle^O,\ \langle\texttt{in}_U\rangle^O,\ \langle\texttt{o}_{\text{OUT}}\rangle^O,\ \langle\texttt{o}_{\text{UNV}}\rangle^O,\ \langle\texttt{o}_\alpha\rangle^O,\ \langle\texttt{o}_{\mathbf{s}}\rangle^O,\ \langle\texttt{o}_{\mathbf{a}}\rangle^O)$ to $f^{(O)}_{\text{OI}}(pid_P,\ k)$
   - Extract pk_U using $\mathsf{POK.ExtractWit}(td_{\text{ext}}, \Pi, stmt, \mathcal{R}_{Val})$
   - Load[a] $pid_{U'} \coloneqq f_{\text{ID}}(\mathsf{pk}_U)$
   - Load current subsession id $ssid$ and append $ssid \mapsto lin^{\text{new}}$ to f_{LN}

   ↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{outsource}k, \texttt{in}_U)$ in the name of U'.
2. Upon receiving output $(\texttt{ok})$ from $\mathcal{F}_{\text{PUBA}}$ to U':
   - $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
   - $(com^{(O)}_{ser^{\text{new}}},\ decom^{(O)}_{ser^{\text{new}}}) \leftarrow \mathsf{COM.commit}((ser^{\text{new}})^{(O)})$
   - $com^{\text{new}}_{\mathcal{UH}} \coloneqq com^{(P)}_{\mathcal{UH}} \oplus com^{(O)}_{\mathcal{UH}}$
   - $com^{\text{new}}_{ser} \coloneqq com^{(U)}_{ser^{\text{new}}} \oplus com^{(O)}_{ser^{\text{new}}}$
   - $(com^{\text{new}}_{lin}, decom^{\text{new}}_{lin}) \leftarrow \mathsf{COM.commit}(lin^{\text{new}})$
   - $\sigma^{\text{new}} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O,\ com^{\text{new}}_{\mathcal{UH}} \| com^{\text{new}}_{ser} \| com^{\text{new}}_{lin} \| \overline{com_{id}})$

↪ Report message $((ser^{\text{new}})^{(O)},\ com^{(O)}_{ser^{\text{new}}},\ decom^{(O)}_{ser^{\text{new}}},\ com^{\text{new}}_{lin},\ decom^{\text{new}}_{lin},\ \sigma^{\text{new}})$ from O to U.
↪ Report message (lin^{new}) from P to U.
↪ Allow $\mathcal{F}_{\text{PUBA}}$ to deliver outputs to P and O.

U honest, P corrupted, O honest:

1. Upon receiving $(\texttt{outsource}k, \text{User})$ from $\mathcal{F}_{\text{PUBA}}$ and $(\texttt{outsource}k, pid_O)$ from $\mathcal{F}_{\text{PUBA}}$:
   - $(\langle \mathcal{UH} \rangle^P, \langle \mathcal{UH} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \texttt{in}_U \rangle^P, \langle \texttt{in}_U \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \texttt{o}_{\text{OUT}} \rangle^P, \langle \texttt{o}_{\text{OUT}} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \texttt{o}_{\text{UNV}} \rangle^P, \langle \texttt{o}_{\text{UNV}} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \texttt{o}_\alpha \rangle^P, \langle \texttt{o}_\alpha \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \texttt{o}_\texttt{s} \rangle^P, \langle \texttt{o}_\texttt{s} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$
   - $(\langle \texttt{o}_\texttt{a} \rangle^P, \langle \texttt{o}_\texttt{a} \rangle^O) \leftarrow \Pi_{\text{Share}}\text{-}Share(\mathbf{0})$

   ↪ Report message $(\langle \mathcal{UH} \rangle^P, \langle \texttt{in}_U \rangle^P, \langle \texttt{o}_{\text{OUT}} \rangle^P, \langle \texttt{o}_{\text{UNV}} \rangle^P, \langle \texttt{o}_\alpha \rangle^P, \langle \texttt{o}_\texttt{s} \rangle^P, \langle \texttt{o}_\texttt{a} \rangle^P)$ from U to P and encrypted message from U to O.
   ↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{outsource}k)$ in the name of P.
2. Upon receiving output $(\texttt{ok})$ from $\mathcal{F}_{\text{PUBA}}$ to P:
   - $(lin^{\text{new}})^O \xleftarrow{r} \mathbb{Z}_p$
   - $(com^{(O)}_{lin^{\text{new}}},\ decom^{(O)}_{lin^{\text{new}}}) \leftarrow \text{COM.commit}((lin^{\text{new}})^O)$
   - Parse:
     - $(*)\mathcal{UH}^{(O)}, com^{(P)}_{\mathcal{UH}}, decom^{(O)}_{\mathcal{UH}} \coloneqq \langle \mathcal{UH} \rangle^O$
     - $(*)\texttt{in}^O_U, com^{(P)}_{\texttt{in}_U}, decom^O_{\texttt{in}_U} \coloneqq \langle \texttt{in}_U \rangle^O$
     - $(*)\texttt{o}_{\text{OUT}}{}^O, com^{(P)}_{\texttt{o}_{\text{OUT}}}, decom^O_{\texttt{o}_{\text{OUT}}} \coloneqq \langle \texttt{o}_{\text{OUT}} \rangle^O$
     - $(*)\texttt{o}_{\text{UNV}}{}^O, com^{(P)}_{\texttt{o}_{\text{UNV}}}, decom^O_{\texttt{o}_{\text{UNV}}} \coloneqq \langle \texttt{o}_{\text{UNV}} \rangle^O$
     - $(*)\texttt{o}_\alpha{}^O, com^{(P)}_{\texttt{o}_\alpha}, decom^O_{\texttt{o}_\alpha} \coloneqq \langle \texttt{o}_\alpha \rangle^O$
     - $(*)\texttt{o}_\texttt{s}{}^O, com^{(P)}_{\texttt{o}_\texttt{s}}, decom^O_{\texttt{o}_\texttt{s}} \coloneqq \langle \texttt{o}_\texttt{s} \rangle^O$
     - $(*)\texttt{o}_\texttt{a}{}^O, com^{(P)}_{\texttt{o}_\texttt{a}}, decom^O_{\texttt{o}_\texttt{a}} \coloneqq \langle \texttt{o}_\texttt{a} \rangle^O$

   ↪ Report message $(com^{(O)}_{lin^{\text{new}}},\ com^{(P)}_{\mathcal{UH}},\ com^{(P)}_{\texttt{in}_U},\ com^{(P)}_{\texttt{o}_{\text{OUT}}},\ com^{(P)}_{\texttt{o}_{\text{UNV}}},\ com^{(P)}_{\texttt{o}_\alpha},\ com^{(P)}_{\texttt{o}_\texttt{s}},\ com^{(P)}_{\texttt{o}_\texttt{a}})$ from O to P.
3. Upon receiving instructions from $\mathcal{Z}$ to send $((lin^{\text{new}})^{(P)},\ com^{(O)}_{\mathcal{UH}},\ com^{(O)}_{\texttt{in}_U},\ com^{(O)}_{\texttt{o}_{\text{OUT}}},\ com^{(O)}_{\texttt{o}_{\text{UNV}}},\ com^{(O)}_{\texttt{o}_\alpha},\ com^{(O)}_{\texttt{o}_\texttt{s}},\ com^{(O)}_{\texttt{o}_\texttt{a}})$ from P to O:
   - Check[a] $\text{open}(com^{(O)}_{\mathcal{UH}}, decom^{(O)}_{\mathcal{UH}}, \mathcal{UH}^{(O)}) \stackrel{?}{=} 1$
   - Check[a] $\text{open}(com^{(O)}_{\texttt{in}_U}, decom^O_{\texttt{in}_U}, \texttt{in}^O_U) \stackrel{?}{=} 1$
   - Check[a] $\text{open}(com^{(O)}_{\texttt{o}_{\text{OUT}}}, decom^O_{\texttt{o}_{\text{OUT}}}, \texttt{o}_{\text{OUT}}{}^O) \stackrel{?}{=} 1$
   - Check[a] $\text{open}(com^{(O)}_{\texttt{o}_{\text{UNV}}}, decom^O_{\texttt{o}_{\text{UNV}}}, \texttt{o}_{\text{UNV}}{}^O) \stackrel{?}{=} 1$
   - Check[a] $\text{open}(com^{(O)}_{\texttt{o}_\alpha}, decom^O_{\texttt{o}_\alpha}, \texttt{o}_\alpha{}^O) \stackrel{?}{=} 1$
   - Check[a] $\text{open}(com^{(O)}_{\texttt{o}_\texttt{s}}, decom^O_{\texttt{o}_\texttt{s}}, \texttt{o}_\texttt{s}{}^O) \stackrel{?}{=} 1$
   - Check[a] $\text{open}(com^{(O)}_{\texttt{o}_\texttt{a}}, decom^O_{\texttt{o}_\texttt{a}}, \texttt{o}_\texttt{a}{}^O) \stackrel{?}{=} 1$

   ↪ Report message $((lin^{\text{new}})^O,\ decom^{(O)}_{lin^{\text{new}}})$ from O to P.
4. Upon receiving instructions from $\mathcal{Z}$ to send (lin^{new}) from P to U:
   - Check[a] $lin^{\text{new}} \stackrel{?}{=} (lin^{\text{new}})^{(O)} + (lin^{\text{new}})^{(P)}$
   - Load current subsession id $ssid$ and append $ssid \mapsto lin^{\text{new}}$ to f_{LN}
   - Append $(lin^{\text{new}}, \langle \mathcal{UH} \rangle^O, \langle \texttt{in}_U \rangle^O, \langle \texttt{o}_{\text{OUT}} \rangle^O, \langle \texttt{o}_{\text{UNV}} \rangle^O, \langle \texttt{o}_\alpha \rangle^O, \langle \texttt{o}_\texttt{s} \rangle^O, \langle \texttt{o}_\texttt{a} \rangle^O)$ to $f^{(O)}_{\text{OI}}(pid_P, k)$

   ↪ Allow $\mathcal{F}_{\text{PUBA}}$ to deliver outputs to U and O.

U and P corrupted, O honest:

1. Upon receiving $(\texttt{outsource}k, pid_O)$ from $\mathcal{F}_{\text{PUBA}}$ and instructions from $\mathcal{Z}$ to send $(\langle \mathcal{UH} \rangle^O, \langle \texttt{in}_U \rangle^O, \langle \texttt{o}_{\text{OUT}} \rangle^O, \langle \texttt{o}_{\text{UNV}} \rangle^O, \langle \texttt{o}_\alpha \rangle^O, \langle \texttt{o}_\texttt{s} \rangle^O, \langle \texttt{o}_\texttt{a} \rangle^O, ser, \overline{com_{id}}, \Pi, com^{(U)}_{ser^{\text{new}}})$ from U to O:
   - Check[a] $ser \notin \mathsf{L}_{\text{SER}}$
   - $\mathsf{L}_{\text{SER}} \coloneqq \mathsf{L}_{\text{SER}} \cup \{ser\}$
   - Parse $\left(\mathcal{UH}^{(O)}, com^{(P)}_{\mathcal{UH}}, decom^{(O)}_{\mathcal{UH}}\right) \coloneqq \langle \mathcal{UH} \rangle^O$
   - Parse $\left(\texttt{in}^O_U, com^{(P)}_{\texttt{in}_U}, decom^O_{\texttt{in}_U}\right) \coloneqq \langle \texttt{in}_U \rangle^O$
   - Parse $\left(\texttt{o}_{\text{OUT}}{}^O, com^{(P)}_{\texttt{o}_{\text{OUT}}}, decom^O_{\texttt{o}_{\text{OUT}}}\right) \coloneqq \langle \texttt{o}_{\text{OUT}} \rangle^O$

- Parse $\left(\mathtt{o}_{\mathrm{UNV}}{}^{O}, com^{(P)}_{\mathtt{o}_{\mathrm{UNV}}}, decom^{O}_{\mathtt{o}_{\mathrm{UNV}}}\right) \coloneqq \langle \mathtt{o}_{\mathrm{UNV}} \rangle^{O}$
- Parse $\left(\mathtt{o}_{\alpha}{}^{O}, com^{(P)}_{\mathtt{o}_{\alpha}}, decom^{O}_{\mathtt{o}_{\alpha}}\right) \coloneqq \langle \mathtt{o}_{\alpha} \rangle^{O}$
- Parse $\left(\mathtt{o}_{\mathtt{s}}{}^{O}, com^{(P)}_{\mathtt{o}_{\mathtt{s}}}, decom^{O}_{\mathtt{o}_{\mathtt{s}}}\right) \coloneqq \langle \mathtt{o}_{\mathtt{s}} \rangle^{O}$
- Parse $\left(\mathtt{o}_{\mathtt{a}}{}^{O}, com^{(P)}_{\mathtt{o}_{\mathtt{a}}}, decom^{O}_{\mathtt{o}_{\mathtt{a}}}\right) \coloneqq \langle \mathtt{o}_{\mathtt{a}} \rangle^{O}$
- $(lin^{\mathrm{new}})^{O} \xleftarrow{r} \mathbb{Z}_p$
- $(com^{(O)}_{lin^{\mathrm{new}}}, decom^{(O)}_{lin^{\mathrm{new}}}) \leftarrow \mathsf{COM.commit}((lin^{\mathrm{new}})^{O})$

↪ Report message $(com^{(O)}_{lin^{\mathrm{new}}}, com^{(P)}_{\mathcal{UH}}, com^{(P)}_{\mathtt{in}_U}, com^{(P)}_{\mathtt{o}_{\mathrm{OUT}}}, com^{(P)}_{\mathtt{o}_{\mathrm{UNV}}}, com_{\mathtt{o}_{\alpha}}, com_{\mathtt{o}_{\mathtt{s}}}, com_{\mathtt{o}_{\mathtt{a}}})$ from O to P

2. Upon receiving instructions from $\mathcal{Z}$ to send $((lin^{\mathrm{new}})^{(P)}, com^{(O)}_{\mathcal{UH}}, com^{(O)}_{\mathtt{in}_U}, com^{(O)}_{\mathtt{o}_{\mathrm{OUT}}}, com^{(O)}_{\mathtt{o}_{\mathrm{UNV}}}, com^{(O)}_{\mathtt{o}_{\alpha}}, com^{(O)}_{\mathtt{o}_{\mathtt{s}}}, com^{(O)}_{\mathtt{o}_{\mathtt{a}}})$ from P to O:
   - $stmt \coloneqq (\langle \mathcal{UH} \rangle^{O}, com^{(O)}_{\mathcal{UH}}, ser, \overline{com_{id}}, \mathsf{vk}_O)$
   - Check[a] $\mathsf{POK.Verify}(crs_{\mathrm{ZK}}, \mathcal{R}_{\mathsf{0S}}, stmt, \Pi) \stackrel{?}{=} 1$
   - Check[a] $\mathsf{open}(com^{(O)}_{\mathcal{UH}}, decom^{(O)}_{\mathcal{UH}}, \mathcal{UH}^{(O)}) \stackrel{?}{=} 1$
   - Check[a] $\mathsf{open}(com^{(O)}_{\mathtt{in}_U}, decom^{O}_{\mathtt{in}_U}, \mathtt{in}^{O}_{U}) \stackrel{?}{=} 1$
   - Check[a] $\mathsf{open}(com^{(O)}_{\mathtt{o}_{\mathrm{OUT}}}, decom^{O}_{\mathtt{o}_{\mathrm{OUT}}}, \mathtt{o}_{\mathrm{OUT}}{}^{O}) \stackrel{?}{=} 1$
   - Check[a] $\mathsf{open}(com^{(O)}_{\mathtt{o}_{\mathrm{UNV}}}, decom^{O}_{\mathtt{o}_{\mathrm{UNV}}}, \mathtt{o}_{\mathrm{UNV}}{}^{O}) \stackrel{?}{=} 1$
   - Check[a] $\mathsf{open}(com^{(O)}_{\mathtt{o}_{\alpha}}, decom^{O}_{\mathtt{o}_{\alpha}}, \mathtt{o}_{\alpha}{}^{O}) \stackrel{?}{=} 1$
   - Check[a] $\mathsf{open}(com^{(O)}_{\mathtt{o}_{\mathtt{s}}}, decom^{O}_{\mathtt{o}_{\mathtt{s}}}, \mathtt{o}_{\mathtt{s}}{}^{O}) \stackrel{?}{=} 1$
   - Check[a] $\mathsf{open}(com^{(O)}_{\mathtt{o}_{\mathtt{a}}}, decom^{O}_{\mathtt{o}_{\mathtt{a}}}, \mathtt{o}_{\mathtt{a}}{}^{O}) \stackrel{?}{=} 1$
   - Extract pk_U using $\mathsf{POK.ExtractWit}(td_{\mathrm{ext}}, \Pi, stmt, \mathcal{R}_{\mathsf{0S}})$
   - Load[a] $pid_{U'} \coloneqq f_{\mathrm{ID}}(\mathsf{pk}_U)$

↪ Call ideal functionality $\mathcal{F}_{\mathrm{PUBA}}$ with input $(\mathtt{outsource}k, \bot)$ in the name of U'

↪ Call ideal functionality $\mathcal{F}_{\mathrm{PUBA}}$ with input $(\mathtt{outsource}k)$ in the name of P

3. Upon receiving output $(\mathtt{ok})$ from $\mathcal{F}_{\mathrm{PUBA}}$ to U' and output $(\mathtt{ok})$ from $\mathcal{F}_{\mathrm{PUBA}}$ to P:
   Compute commitments and signature:
   - Set $lin \coloneqq (lin^{\mathrm{new}})^{O} + (lin^{\mathrm{new}})^{(P)}$
   - Append $(lin^{\mathrm{new}}, \langle \mathcal{UH} \rangle^{O}, \langle \mathtt{in}_U \rangle^{O}, \langle \mathtt{o}_{\mathrm{OUT}} \rangle^{O}, \langle \mathtt{o}_{\mathrm{UNV}} \rangle^{O}, \langle \mathtt{o}_{\alpha} \rangle^{O}, \langle \mathtt{o}_{\mathtt{s}} \rangle^{O}, \langle \mathtt{o}_{\mathtt{a}} \rangle^{O})$ to $f^{(O)}_{\mathrm{OI}}(pid_P, k)$
   - $(ser^{\mathrm{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
   - $(com^{(O)}_{ser^{\mathrm{new}}}, decom^{(O)}_{ser^{\mathrm{new}}}) \leftarrow \mathsf{COM.commit}((ser^{\mathrm{new}})^{(O)})$
   - $com^{\mathrm{new}}_{\mathcal{UH}} \coloneqq com^{(P)}_{\mathcal{UH}} \oplus com^{(O)}_{\mathcal{UH}}$
   - $com^{\mathrm{new}}_{ser} \coloneqq com^{(U)}_{ser^{\mathrm{new}}} \oplus com^{(O)}_{ser^{\mathrm{new}}}$
   - $(com^{\mathrm{new}}_{lin}, decom^{\mathrm{new}}_{lin}) \leftarrow \mathsf{COM.commit}(lin^{\mathrm{new}})$
   - $\sigma^{\mathrm{new}} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O, com^{\mathrm{new}}_{\mathcal{UH}} \| com^{\mathrm{new}}_{ser} \| com^{\mathrm{new}}_{lin} \| \overline{com_{id}})$

   ↪ Report message $((lin^{\mathrm{new}})^{(O)}, decom^{(O)}_{lin^{\mathrm{new}}})$ from O to P and message $((ser^{\mathrm{new}})^{(O)}, com^{(O)}_{ser^{\mathrm{new}}}, decom^{(O)}_{ser^{\mathrm{new}}}, com^{\mathrm{new}}_{lin}, decom^{\mathrm{new}}_{lin}, \sigma^{\mathrm{new}})$ from O to U.

   ↪ Allow $\mathcal{F}_{\mathrm{PUBA}}$ to deliver outputs to O.

$\{\Pi_{\mathrm{PUBA}}\text{-}Analytics^{(k)}\}^{K}_{k=1}$:

P and O honest:

1. Upon receiving $(\mathtt{analyt}k, pid_P)$ from $\mathcal{F}_{\mathrm{PUBA}}$ and $(\mathtt{analyt}k, pid_O)$ from $\mathcal{F}_{\mathrm{PUBA}}$:
   - Load[a] and remove the first Z_k entries $\{(lin, \langle \mathcal{UH} \rangle^{P}, \langle \mathtt{in}_U \rangle^{P}, \langle \mathtt{o}_{\mathrm{OUT}} \rangle^{P}, \langle \mathtt{o}_{\mathrm{UNV}} \rangle^{P}, \langle \mathtt{o}_{\alpha} \rangle^{P}, \langle \mathtt{o}_{\mathtt{s}} \rangle^{P}, \langle \mathtt{o}_{\mathtt{a}} \rangle^{P})_z\}^{Z_k}_{z=1}$ from $f^{(P)}_{\mathrm{OI}}(k)$
   - Load[a] and remove the first Z_k entries $\{(lin, \langle \mathcal{UH} \rangle^{O}, \langle \mathtt{in}_U \rangle^{O}, \langle \mathtt{o}_{\mathrm{OUT}} \rangle^{O}, \langle \mathtt{o}_{\mathrm{UNV}} \rangle^{O}, \langle \mathtt{o}_{\alpha} \rangle^{O}, \langle \mathtt{o}_{\mathtt{s}} \rangle^{O}, \langle \mathtt{o}_{\mathtt{a}} \rangle^{O})_z\}^{Z_k}_{z=1}$ from $f^{(O)}_{\mathrm{OI}}(pid_P, k)$
   - For every $z \in \{1, \ldots, Z_k\}$ with $lin_z \neq \bot$, combine[a]:
     - $\mathcal{UH}_z \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle \mathcal{UH}_z \rangle^{P}, \langle \mathcal{UH}_z \rangle^{O})$
     - $\mathtt{in}_{U_z} \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle \mathtt{in}_{U_z} \rangle^{P}, \langle \mathtt{in}_{U_z} \rangle^{O})$
     - $\mathtt{o}_{\mathrm{OUT}\,z} \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle \mathtt{o}_{\mathrm{OUT}\,z} \rangle^{P}, \langle \mathtt{o}_{\mathrm{OUT}\,z} \rangle^{O})$
     - $\mathtt{o}_{\mathrm{UNV}\,z} \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle \mathtt{o}_{\mathrm{UNV}\,z} \rangle^{P}, \langle \mathtt{o}_{\mathrm{UNV}\,z} \rangle^{O})$
     - $\mathtt{o}_{\alpha\,z} \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle \mathtt{o}_{\alpha\,z} \rangle^{P}, \langle \mathtt{o}_{\alpha\,z} \rangle^{O})$
     - $\mathtt{o}_{\mathtt{s}\,z} \coloneqq \Pi_{\mathrm{Share}}\text{-}Combine(\langle \mathtt{o}_{\mathtt{s}\,z} \rangle^{P}, \langle \mathtt{o}_{\mathtt{s}\,z} \rangle^{O})$

- $\mathrm{o}_{\mathbf{a}z} \coloneqq \Pi_{\text{Share}}\text{-}Combine(\langle \mathrm{o}_{\mathbf{a}z}\rangle^P, \langle \mathrm{o}_{\mathbf{a}z}\rangle^O)$

↪ Allow $\mathcal{F}_{\text{PUBA}}$ to continue.

2. Upon receiving leak $(z, \alpha_z, \mathbf{s}_z, \mathbf{a}_z, \mathtt{out}_{U_z})$ from $\mathcal{F}_{\text{PUBA}}$, do for each z for which a leak $(z, \cdot, \cdot, \cdot, \cdot)$ exists:
   - $(com_{\mathbf{a}_z}, decom_{\mathbf{a}_z}) \xleftarrow{r} \mathsf{commit}(\mathbf{a}_z)$
   - $ct_{\alpha_z} \coloneqq \alpha_z + \mathrm{o}_{\alpha z}$
   - $ct_{\mathbf{s}_z} \coloneqq \mathbf{s}_z + \mathrm{o}_{\mathbf{s}z}$
   - $ct_{\mathbf{a}_z} \coloneqq \mathbf{a}_z + \mathrm{o}_{\mathbf{a}z}$
   - $ct_{decom_{\mathbf{a}_z}} \coloneqq decom_{\mathbf{a}_z} + \mathrm{o}_{\text{UNV}}$
   - $ct_{\mathtt{out}_{U_z}} \coloneqq \mathtt{out}_{U_z} + \mathrm{o}_{\text{OUT}z}$
   - $f_{\text{UI}}^{(P)}(lin_z) \coloneqq (ct_\alpha, ct_{\mathbf{s}}, ct_{\mathbf{a}}, ct_{decom_{\mathbf{a}}}, ct_{\mathtt{out}_U})_z$
   - $f_{\text{UI}}^{(O)}(lin_z) \coloneqq (\alpha_z, \mathbf{s}_z, com_{\mathbf{a}_z}, ct_{\mathtt{out}_U})_z$

↪ Allow $\mathcal{F}_{\text{PUBA}}$ to deliver outputs to all parties.

P corrupted, O honest:

1. Upon receiving $(\mathtt{analyt}k, pid_O)$ from $\mathcal{F}_{\text{PUBA}}$ and leak ℓ from $\mathcal{F}_{\text{PUBA}}$:
   - $(com_0, decom_0, \sigma_{fp}) \leftarrow f_{\text{FP}}(\mathtt{analyt}k, \ell)$

↪ Report message (com_0, σ_{fp}) from O to P.

2. Upon receiving instructions from $\mathcal{Z}$ to send $(\mathtt{analyt}k, com_{fp}, \{(\langle \mathcal{UH}\rangle^P, \langle \mathtt{in}_U\rangle^P, \langle \mathrm{o}_{\text{OUT}}\rangle^P, \langle \mathrm{o}_{\text{UNV}}\rangle^P, \langle \mathrm{o}_\alpha\rangle^P, \langle \mathrm{o}_{\mathbf{s}}\rangle^P, \langle \mathrm{o}_{\mathbf{a}}\rangle^P)_z\}_{z=1}^{Z_k}$ from P to $\mathcal{F}_{\text{PPA}}$:
   - Check[a] $\mathsf{COM.open}(com_{fp}, decom_0, \mathbf{0}) = 1$.
   - Load[a] and remove the first Z_k entries $\{(lin, \langle \mathcal{UH}\rangle^O, \langle \mathtt{in}_U\rangle^O, \langle \mathrm{o}_{\text{OUT}}\rangle^O, \langle \mathrm{o}_{\text{UNV}}\rangle^O, \langle \mathrm{o}_\alpha\rangle^O, \langle \mathrm{o}_{\mathbf{s}}\rangle^O, \langle \mathrm{o}_{\mathbf{a}}\rangle^O)_z\}_{z=1}^{Z_k}$ from $f_{\text{OI}}^{(O)}(pid_P, k)$
   - For every z from 1 to Z_k, combine[a]:
     - $\mathcal{UH}_z \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \mathcal{UH}_z\rangle^P, \langle \mathcal{UH}_z\rangle^O)$
     - $\mathtt{in}_{U_z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \mathtt{in}_z\rangle^P, \langle \mathtt{in}_z\rangle^O)$
     - $\mathrm{o}_{\text{OUT}z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \mathrm{o}_{\text{OUT}z}\rangle^P, \langle \mathrm{o}_{\text{OUT}z}\rangle^O)$
     - $\mathrm{o}_{\text{UNV}z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \mathrm{o}_{\text{UNV}z}\rangle^P, \langle \mathrm{o}_{\text{UNV}z}\rangle^O)$
     - $\mathrm{o}_{\alpha z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \mathrm{o}_{\alpha z}\rangle^P, \langle \mathrm{o}_{\alpha z}\rangle^O)$
     - $\mathrm{o}_{\mathbf{s}z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \mathrm{o}_{\mathbf{s}z}\rangle^P, \langle \mathrm{o}_{\mathbf{s}z}\rangle^O)$
     - $\mathrm{o}_{\mathbf{a}z} \leftarrow \Pi_{\text{Share}}\text{-}Combine(\langle \mathrm{o}_{\mathbf{a}z}\rangle^P, \langle \mathrm{o}_{\mathbf{a}z}\rangle^O)$

↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\mathtt{analyt}k)$ in the name of P.

3. Upon being asked by $\mathcal{F}_{\text{PUBA}}$ for updated inputs for a set $id_{\text{corrupted}} \coloneqq \{[Z_k], U_z \text{ corrupted}\}$:
   - Send inputs $\{\mathtt{in}_{U_z} | z \in id_{\text{corrupted}}\}$ to $\mathcal{F}_{\text{PUBA}}$

↪ Allow $\mathcal{F}_{\text{PUBA}}$ to continue.

4. Upon receiving leak $\{(z, \alpha_z, \mathbf{s}_z, \mathbf{a}_z, \mathtt{out}_{U_z}) | z \in id_{\text{corrupted}}\}$ from $\mathcal{F}_{\text{PUBA}}$ and output $(\mathtt{ok})$ from $\mathcal{F}_{\text{PUBA}}$ to P:
   - For each z in $1, \ldots, Z_k$:
     - If an entry $\{(z), \cdot, \cdot, \cdot, \cdot\}$ exists in the leaked set:
       - $ct_{\alpha_z} \coloneqq \alpha_z + \mathrm{o}_{\alpha z}$
       - $ct_{\mathbf{s}_z} \coloneqq \mathbf{s}_z + \mathrm{o}_{\mathbf{s}z}$
       - $ct_{\mathbf{a}_z} \coloneqq \mathbf{a}_z + \mathrm{o}_{\mathbf{a}z}$
       - $(com_{\mathbf{a}_z}, decom_{\mathbf{a}_z}) \leftarrow \mathsf{commit}(\mathbf{a}_z)$
       - $ct_{decom_{\mathbf{a}_z}} \coloneqq decom_{\mathbf{a}_z} + \mathrm{o}_{\text{UNV}z}$
       - $ct_{\mathtt{out}_{U_z}} \coloneqq \mathtt{out}_{U_z} + \mathrm{o}_{\text{OUT}z}$
       - $f_{\text{UI}}^{(P)}(lin_z) \coloneqq (ct_{\alpha_z}, ct_{\mathbf{s}_z}, ct_{\mathbf{a}_z}, ct_{decom_{\mathbf{a}_z}}, ct_{\mathtt{out}_{U_z}})$
       - $f_{\text{UI}}^{(O)}(lin_z) \coloneqq (\alpha, \mathbf{s}, com_{\mathbf{a}_z}, ct_{\mathtt{out}_{U_z}})$
     - Else:
       - Draw random $ct_{\alpha_z}, ct_{\mathbf{s}_z}, ct_{\mathbf{a}_z}, ct_{\mathtt{out}_{U_z}}$ and $ct_{decom_{\mathbf{a}_z}}$
       - $f_{\text{UI}}^{(P)}(lin_z) \coloneqq (ct_{\alpha_z}, ct_{\mathbf{s}_z}, ct_{\mathbf{a}_z}, ct_{decom_{\mathbf{a}_z}}, ct_{\mathtt{out}_{U_z}})$

↪ Report message $\{(ct_{\alpha_z}, ct_{\mathbf{s}_z}, ct_{\mathbf{a}_z}, ct_{decom_{\mathbf{a}_z}}, ct_{\mathtt{out}_{U_z}})\}_{z=1}^{Z_k}$ from $\mathcal{F}_{\text{PPA}}$ to P.

↪ Allow $\mathcal{F}_{\text{PUBA}}$ to deliver outputs to O.

Π_{PUBA}-*Update*:

U, P and O honest: Report encrypted messages.

U corrupted, P and O honest:

1. Upon receiving $(\texttt{update}, pid_P)$ from $\mathcal{F}_{\text{PUBA}}$ and instructions from $\mathcal{Z}$ to send (lin) from U to P:
   - If $f_{\text{UI}}^{(P)}(lin) \neq \bot$:
     - Load and remove $(ct_\alpha,\ ct_{\mathbf{s}},\ ct_{\mathbf{a}},\ ct_{decom_{\mathbf{a}}},\ ct_{\texttt{out}_U}) \coloneqq f_{\text{UI}}^{(P)}(lin)$

   $\hookrightarrow$ Report message $(ct_\alpha,\ ct_{\mathbf{s}},\ ct_{\mathbf{a}},\ ct_{decom_{\mathbf{a}}},\ ct_{\texttt{out}_U})$ from P to U.
   - otherwise continue without reporting a message
2. Upon receiving $(\texttt{update}, pid_O)$ from $\mathcal{F}_{\text{PUBA}}$:
   - Load[a] and remove $(\alpha,\ \mathbf{s},\ com_{\mathbf{a}},\ ct_{\texttt{out}_U}) \coloneqq f_{\text{UI}}^{(O)}(lin)$
3. **if** $\alpha \neq \bot \vee \mathbf{s} \neq \bot$ **then**

   Upon receiving instructions from $\mathcal{Z}$ to send $(\overline{com_{\mathcal{UH}}},\ com'_{\mathcal{UH}},\ com''_{\mathcal{UH}},\ \overline{com_{id}},\ ser,\ lin,\ \Pi,\ \Pi_{Tr},\ com_{ser^{\text{new}}}^{(U)})$ from U to O:

   Check proof:
   - $stmt \coloneqq (\overline{com_{\mathcal{UH}}},\ ser,\ lin,\ \overline{com_{id}},\ \mathsf{vk}_O)$
   - $stmt_{Tr} \coloneqq (\overline{com_{\mathcal{UH}}},\ com'_{\mathcal{UH}},\ com''_{\mathcal{UH}},\ \alpha,\ \mathbf{s})$
   - Check[a] $\text{POK.Verify}(\Pi,\ stmt,\ \mathcal{R}_{Upd}) \stackrel{?}{=} 1$
   - Check[a] $\text{POK.Verify}(crs_{\text{ZK}},\ \mathcal{R}_{Tr},\ stmt_{Tr},\ \Pi_{Tr}) \stackrel{?}{=} 1$
   - $com_{\mathcal{UH}}^{\text{new}} \coloneqq com''_{\mathcal{UH}} \oplus com_{\mathbf{a}}$
4. **else**

   Upon receiving instructions from $\mathcal{Z}$ to send $(\overline{com_{\mathcal{UH}}},\ \overline{com_{id}},\ ser,\ lin,\ \Pi,\ com_{ser^{\text{new}}}^{(U)})$ from U to O:

   Check proof:
   - $stmt \coloneqq (\overline{com_{\mathcal{UH}}},\ ser,\ lin,\ \overline{com_{id}},\ \mathsf{vk}_O)$
   - Check[a] $\text{POK.Verify}(\Pi,\ stmt,\ \mathcal{R}_{Upd}) \stackrel{?}{=} 1$
   - $com_{\mathcal{UH}}^{\text{new}} \coloneqq \overline{com_{\mathcal{UH}}} \oplus com_{\mathbf{a}}$
5. **fi**

   Check serial number:
   - Check[a] $ser \notin \mathsf{L}_{\text{SER}}$
   - $\mathsf{L}_{\text{SER}} \coloneqq \mathsf{L}_{\text{SER}} \cup \{ser\}$
   - Extract pk_U using $\text{POK.ExtractWit}(td_{\text{ext}},\ \Pi,\ stmt,\ \mathcal{R}_{Upd})$
   - Load[a] $pid_{U'} \coloneqq f_{\text{ID}}(\mathsf{pk}_U)$

   $\hookrightarrow$ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{update})$ in the name of U'.
6. Upon receiving output $(\alpha,\ \mathbf{s},\ \mathbf{a},\ \texttt{out}_U)$ from $\mathcal{F}_{\text{PUBA}}$ to U':

   Draw share of new serial number:
   - $(ser^{\text{new}})^{(O)} \stackrel{\text{r}}{\leftarrow} \mathbb{Z}_p$
   - $(com_{ser^{\text{new}}}^{(O)},\ decom_{ser^{\text{new}}}^{(O)}) \leftarrow \text{COM.commit}((ser^{\text{new}})^{(O)})$

   Compute commitments and signature for updated user history:
   - $com_{ser}^{\text{new}} \coloneqq com_{ser^{\text{new}}}^{(O)} \oplus com_{ser^{\text{new}}}^{(U)}$
   - $(com_{lin}^{\text{new}},\ decom_{lin}^{\text{new}}) \leftarrow \text{COM.commit}(0)$
   - $\sigma^{\text{new}} \leftarrow \text{SIG.Sign}(\mathsf{sk}_O,\ com_{\mathcal{UH}}^{\text{new}} \| com_{ser}^{\text{new}} \| com_{lin}^{\text{new}} \| \overline{com_{id}})$

   $\hookrightarrow$ Report message $(com_{\mathbf{a}},\ com_{\mathcal{UH}}^{\text{new}},\ (ser^{\text{new}})^{(O)},\ com_{ser^{\text{new}}}^{(O)},\ decom_{ser^{\text{new}}}^{(O)},\ com_{lin}^{\text{new}},\ decom_{lin}^{\text{new}},\ \sigma^{\text{new}},\ ct_{\texttt{out}_U})$ from O to U.

   $\hookrightarrow$ Allow $\mathcal{F}_{\text{PUBA}}$ to deliver outputs.

U honest, P corrupted, O honest:

1. Upon receiving $(\texttt{update}, pid_O)$ and $(\texttt{update}, \text{User})$ from $\mathcal{F}_{\text{PUBA}}$:

   $\hookrightarrow$ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{update})$ in the name of P.
2. Upon receiving leak $ssid$ from $\mathcal{F}_{\text{PUBA}}$:
   - Load[a] $lin \coloneqq f_{\text{LN}}(ssid)$

   $\hookrightarrow$ Report message (lin) from U to P.
3. Upon receiving instructions from $\mathcal{Z}$ to send $(ct_\alpha,\ ct_{\mathbf{s}},\ ct_{\mathbf{a}},\ ct_{decom_{\mathbf{a}}},\ ct_{\texttt{out}_U})$ from P to U:
   - Check[a] $(ct_\alpha,\ ct_{\mathbf{s}},\ ct_{\mathbf{a}},\ ct_{decom_{\mathbf{a}}},\ ct_{\texttt{out}_U}) \stackrel{?}{=} f_{\text{UI}}^{(P)}(lin)$ and remove the entry.

   $\hookrightarrow$ Allow $\mathcal{F}_{\text{PUBA}}$ to continue.

U and P corrupted, O honest:

1. Upon receiving $(\texttt{update}, pid_O)$ from $\mathcal{F}_{\text{PUBA}}$:
   - Load[a] and remove $(\alpha,\ \mathbf{s},\ com_{\mathbf{a}},\ ct_{\text{out}_U}) \coloneqq f_{\text{UI}}^{(O)}(lin)$.
2. **if** $\alpha \neq \bot \vee \mathbf{s} \neq \bot$ **then**
   Upon receiving instructions from $\mathcal{Z}$ to send $(\overline{com_{\mathcal{UH}}},\ com'_{\mathcal{UH}},\ com''_{\mathcal{UH}},\ \overline{com_{id}},\ ser,\ lin,\ \Pi,\ \Pi_{Tr},\ com_{ser^{\text{new}}}^{(U)})$ from U to O:
   Check proof:
   - $stmt \coloneqq (\overline{com_{\mathcal{UH}}},\ ser,\ lin,\ \overline{com_{id}},\ \text{vk}_O)$
   - $stmt_{Tr} \coloneqq (\overline{com_{\mathcal{UH}}},\ com'_{\mathcal{UH}},\ com''_{\mathcal{UH}},\ \alpha,\ \mathbf{s})$
   - Check[a] $\text{POK.Verify}(\Pi,\ stmt,\ \mathcal{R}_{Upd}) \stackrel{?}{=} 1$
   - Check[a] $\text{POK.Verify}(crs_{\text{ZK}},\ \mathcal{R}_{Tr},\ stmt_{Tr},\ \Pi_{Tr}) \stackrel{?}{=} 1$
   - $com^{\text{new}}_{\mathcal{UH}} \coloneqq com''_{\mathcal{UH}} \oplus com_{\mathbf{a}}$
3. **else**
   Upon receiving instructions from $\mathcal{Z}$ to send $(\overline{com_{\mathcal{UH}}},\ \overline{com_{id}},\ ser,\ lin,\ \Pi,\ com_{ser^{\text{new}}}^{(U)})$ from U to O:
   Check proof:
   - $stmt \coloneqq (\overline{com_{\mathcal{UH}}},\ ser,\ lin,\ \overline{com_{id}},\ \text{vk}_O)$
   - Check[a] $\text{POK.Verify}(\Pi,\ stmt,\ \mathcal{R}_{Upd}) \stackrel{?}{=} 1$
   - $com^{\text{new}}_{\mathcal{UH}} \coloneqq \overline{com_{\mathcal{UH}}} \oplus com_{\mathbf{a}}$
4. **fi**
   Check serial number:
   - Check[a] $ser \notin \mathsf{L}_{\text{SER}}$
   - $\mathsf{L}_{\text{SER}} \coloneqq \mathsf{L}_{\text{SER}} \cup \{ser\}$
   - Extract pk_U using $\text{POK.ExtractWit}(td_{\text{ext}}, \Pi, stmt, \mathcal{R}_{Upd})$
   - Load[a] $pid_{U'} \coloneqq f_{\text{ID}}(\text{pk}_U)$

   ↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{update})$ in the name of U'.
   ↪ Call ideal functionality $\mathcal{F}_{\text{PUBA}}$ with input $(\texttt{update})$ in the name of P.
5. Upon receiving output $(\texttt{ok})$ from $\mathcal{F}_{\text{PUBA}}$ to P and output $(\alpha, \mathbf{s}, \mathbf{a}, \text{out}_U)$ from $\mathcal{F}_{\text{PUBA}}$ to U':
   - $(ser^{\text{new}})^{(O)} \xleftarrow{r} \mathbb{Z}_p$
   - $(com_{ser^{\text{new}}}^{(O)},\ decom_{ser^{\text{new}}}^{(O)}) \leftarrow \text{COM.commit}((ser^{\text{new}})^{(O)})$
   - $com^{\text{new}}_{ser} \coloneqq com_{ser^{\text{new}}}^{(O)} \oplus com_{ser^{\text{new}}}^{(U)}$
   - $(com^{\text{new}}_{lin},\ decom^{\text{new}}_{lin}) \leftarrow \text{COM.commit}(0)$
   - $\sigma^{\text{new}} \leftarrow \text{SIG.Sign}(\text{sk}_O,\ com^{\text{new}}_{\mathcal{UH}} \| com^{\text{new}}_{ser} \| com^{\text{new}}_{lin} \| \overline{com_{id}})$

   ↪ Report message $(com_{\mathbf{a}},\ com^{\text{new}}_{\mathcal{UH}},\ (ser^{\text{new}})^{(O)},\ com_{ser^{\text{new}}}^{(O)},\ decom_{ser^{\text{new}}}^{(O)},\ com^{\text{new}}_{lin},\ decom^{\text{new}}_{lin},\ \sigma^{\text{new}},\ ct_{\text{out}_U})$ from O to U.
   ↪ Allow $\mathcal{F}_{\text{PUBA}}$ to deliver output to O.

[a] If this fails, output $\bot$ and abort.
[b] If this fails, use a default value $\mathbf{0}$.

For our proof, we consider the following hybrid games H_i:

H_1 The hybrid H_1 is equivalent to the real experiment. That is,

$$\mathsf{H}_1 \coloneqq \text{EXEC}_{\Pi_{\text{PUBA}}, \mathcal{F}_{\text{BB}}, \mathcal{F}_{\text{PPA}}, \mathcal{S}_1, \mathcal{Z}}(1^\lambda)$$

This means that all parties execute the real protocol.

H_2 All calls to hybrid functionalities, namely to $\mathcal{F}_{\text{PPA}}$, $\mathcal{F}_{\text{KE}}$ and $\mathcal{F}_{\text{BB}}$, are replaced by calls to $\mathcal{S}_2$, who simulates their behavior using the respective code of the simulators $\mathcal{S}_{\mathcal{F}_{\text{PPA}}}$ and $\mathcal{S}_{\mathcal{F}_{\text{BB}}}$.

H_3 The simulator now maintains the list L_{SER}, in which it stores the serials that were opened by a user. That is, whenever a user proves that the used logbook Log is "fresh" and hasn't been used before by sending a serial number *ser* together with a proof Π during any of the tasks for Bookkeeping, Analytics or Update, $\mathcal{S}_3$ verifies the proof and aborts, if either the proof fails to verify or if serial is already contained

in $\mathsf{L}_{\mathsf{SER}}$. If no abort happened, $\mathcal{S}_3$ adds *ser* to the list $\mathsf{L}_{\mathsf{SER}}$. The code of the operator O is changed in such a way, that the checks that the simulator does now are not performed again by the operator.

H_4 Introduces a map $f_{\mathrm{UI}}^{(O)}$ and for each proxy a map $f_{\mathrm{UI}}^{(P)}$ to be used by the simulator $\mathcal{S}_4$: $f_{\mathrm{UI}}^{(O)}$ is similar to what an honest operator would store for the updates after the Analytics-task, $f_{\mathrm{UI}}^{(P)}$ is the equivalent for the respective proxy. During simulation of $\mathcal{F}_{\mathrm{PPA}}$ for Analytics-tasks, i.e. after O sent a message $(\texttt{analyt}k,\ fp,\ decom_{fp},\ \{(\langle\mathcal{UH}\rangle^O,\ \langle\texttt{in}_U\rangle^O,\ \langle\mathsf{o}_{\mathrm{OUT}}\rangle^O,\ \langle\mathsf{o}_{\mathrm{UNV}}\rangle^O,\ \langle\mathsf{o}_\alpha\rangle^O,\ \langle\mathsf{o}_{\mathbf{s}}\rangle^O,\ \langle\mathsf{o}_{\mathbf{a}}\rangle^O)_z\}_{z=1}^{Z_k},\ \texttt{in}_O)$ and P input $(\texttt{analyt}k,\ com_{fp},\ \{(\langle\mathcal{UH}\rangle^P,\ \langle\texttt{in}_U\rangle^P,\ \langle\mathsf{o}_{\mathrm{OUT}}\rangle^P,\ \langle\mathsf{o}_{\mathrm{UNV}}\rangle^P,\ \langle\mathsf{o}_\alpha\rangle^P,\ \langle\mathsf{o}_{\mathbf{s}}\rangle^P,\ \langle\mathsf{o}_{\mathbf{a}}\rangle^P)_z\}_{z=1}^{Z_k})$ to $\mathcal{F}_{\mathrm{PPA}}$, Simulator $\mathcal{S}_4$ computes Δ honestly (with fresh coins, if necessary), based on the two inputs. $\mathcal{S}_4$ uses the protocol Π_{Share}-*Combine* on both received shares to reconstruct $\mathcal{UH}_z$, $\mathsf{o}_{\alpha z}$, $\mathsf{o}_{\mathbf{s}z}$, $\mathsf{o}_{\mathbf{a}z}$, $\mathsf{o}_{\mathrm{OUT}z}$ and $\mathsf{o}_{\mathrm{UNV}z}$ for each user $z \in [Z]$. If reconstruction on any of the shares fails, $\mathcal{S}_4$ aborts. With this, $\mathcal{S}_4$ computes $ct_{\alpha_z} \coloneqq \alpha_z + \mathsf{o}_{\alpha z}$, $ct_{\mathbf{s}_z} \coloneqq \mathbf{s}_z + \mathsf{o}_{\mathbf{s}z}$, $ct_{\mathbf{a}_z} \coloneqq \mathbf{a}_z + \mathsf{o}_{\mathbf{a}z}$, $(com_{\mathbf{a}_z},\ decom_{\mathbf{a}_z}) \leftarrow \mathsf{COM.commit}(\mathbf{a}_z)$, $ct_{decom_{\mathbf{a}_z}} \coloneqq decom_{\mathbf{a}_z} + \mathsf{o}_{\mathrm{UNV}}$ and $ct_{\mathtt{out}_z} \coloneqq \mathtt{out}_z + \mathsf{o}_{\mathrm{OUT}z}$. $\mathcal{S}_4$ then adds a new entry $(lin_z \mapsto (ct_\alpha,\ ct_{\mathbf{s}},\ ct_{\mathbf{a}},\ ct_{decom_{\mathbf{a}}, ct_{\mathtt{out}_U}}))$ to $f_{\mathrm{UI}}^{(P)}$ and a new entry $(lin_z \mapsto (\alpha,\ \mathbf{s},\ com_{\mathbf{a}},\ ct_{\mathtt{out}_U}))$ to $f_{\mathrm{UI}}^{(O)}$.

$\mathcal{S}_4$ also uses $f_{\mathrm{UI}}^{(P)}$ to verify that a corrupted P sent the correct values during the Update-task to an honest U, which replaces Us check with the one-time pads.

H_5 During setup, the reference string *crs* is created by $(crs_{\mathrm{ZK}},\ td_{\mathrm{ext}}) \leftarrow \mathsf{SetupEPoK}$. Also, the operator now leaks the signature key pair $(\mathsf{vk}_O, \mathsf{sk}_O)$ to $\mathcal{S}_5$ during the Init-task; the simulator then stores $(\mathsf{vk}_O, \mathsf{sk}_O)$. The simulator stores td_{ext}, the remainder stays as it is.

H_6 Introduces a new map f_{ID} for the simulator that uniquely maps user's public keys pk_U to the pid pid_U the respective user had during user registration.

During simulation of $\mathcal{F}_{\mathrm{BB}}$, after $\mathcal{Z}$ gave instructions to send a message $(\texttt{Register},\ pid_U,\ \mathsf{pk}_U)$ from a corrupted user U to $\mathcal{F}_{\mathrm{BB}}$ and simulation succeeded (i.e. did not abort), $\mathcal{S}_6$ adds a new entry $(\mathsf{pk}_U \mapsto pid_U)$ to f_{ID}.

During simulation UReg, after $\mathcal{Z}$ gave instructions to send a message $(\Pi,\ com_{id},\ com_{ser^{\mathrm{new}}}^{(U)})$ from a corrupted U to O, $\mathcal{S}_6$ takes pk'_U from Π (since it is contained in the statement of $\mathcal{R}_{\mathsf{UR}}$) and aborts if either $f_{\mathrm{ID}}(\mathsf{pk}'_U) \neq pid_U$ or if a user with public key pk'_U is already registered.

H_7 Whenever $\mathcal{Z}$ instructs $\mathcal{S}_7$ to send a message containing a zero-knowledge proof Π in the name of a corrupted user U during the tasks for Bookkeeping, Analytics and Update, $\mathcal{S}_7$ uses the trapdoor td_{ext} to extract the complete witness *wit* from Π. It then uses the extracted public key pk_U to get the pid $pid_U \coloneqq f_{\mathrm{ID}}(\mathsf{pk}_U)$ of the user whom the user history belongs to and aborts if $pid_U = \perp$.

H_8 Introduces an incorruptible entity $\mathcal{F}_{\text{PUBA}}$ that follows the specification from Section A.3 into the experiment, which is only accessible by honest parties and the simulator through subroutine input/output tapes.

H_9 Replaces the trusted signing authority $\mathcal{TSA}$ with a dummy party that immediately forwards its input to the ideal functionality $\mathcal{F}_{\text{PUBA}}$. All interactions of $\mathcal{TSA}$ are simulated by $\mathcal{S}_9$ by following the original protocol.

H_{10} Introduces a map f_{FP} that maps a given task $\texttt{task} \in \{\texttt{ureg} \cup \texttt{bookkeep}K \cup \texttt{analyt}K\}$ and a given identifier $\ell \in [L]$ to a tuple $(fp,\ com_{fp},\ decom_{fp},\ \sigma_{fp})$.

Replaces the operator O with a new operator O' that acts like the original one, but has a few minor changes. After having confirmation that fp can be used for a task $\texttt{task}$ during SignFP, O' sends $(fp,\ \texttt{task})$ to $\mathcal{S}_{10}$, who computes com_{fp} as commitment on the actual function parameters and σ_{fp} as corresponding signatures honestly and stores it in $f_{\text{FP}}(fp,\ com_{fp},\ \sigma_{fp})$. Furthermore, the new operator sends the fp to $\mathcal{S}_{10}$ during UReg, Bookkeeping and Analytics tasks and uses the $(com_{fp},\ \sigma_{fp})$ obtained from $\mathcal{S}_{10}$, which the simulator obtains by looking if an entry $(fp,\ \cdot,\ \cdot)$ exists within f_{FP}.

Also, during simulation of $\mathcal{F}_{\text{PPA}}$, $\mathcal{S}_{10}$ uses the decommitment information stored in $f_{\text{FP}}(\texttt{task},\ \ell)$ to verify the commitment com_{fp} the user or proxy input into $\mathcal{F}_{\text{PPA}}$.

H_{11} Replaces the operator O by a dummy party, which, when receiving input by $\mathcal{Z}$, forwards the input immediately to $\mathcal{F}_{\text{PUBA}}$. During the tasks for UReg, Bookkeeping and Analytics, the new operator O also leaks the input $\texttt{in}_O$ and fp to $\mathcal{S}_{11}$. All messages that were sent by O in H_{10} are now created by $\mathcal{S}_{11}$ in H_{11} and send in the name of O.

H_{12} Replaces the map f_{FP} introduced in H_{10} with one that has the same input space $(\texttt{task},\ \ell)$, but whose output space only contains $(com_{fp},\ decom_{fp},\ \sigma_{fp})$, and not the actual function parameters.

Instead of relying on the leaked fp for a mapping, $\mathcal{S}_{12}$ uses the leak ℓ obtained from $\mathcal{F}_{\text{PUBA}}$ to obtain a consistent $(com_{fp},\ \sigma_{fp})$ tuple. Furthermore, instead of creating these tuples honestly, $\mathcal{S}_{12}$ uses the leaked $(\texttt{task},\ \ell)$ obtained during the SignFP task to sample a new $(com_{\mathbf{0}},\ decom_{\mathbf{0}}) \leftarrow \text{COM.commit}(\mathbf{0})$ on the all-zero vector $\mathbf{0}$ instead of com_{fp} on fp, and uses the signing key $\text{sk}_{\mathcal{TSA}}$ to sign $com_{\mathbf{0}}$ and the given task $\texttt{task}$. The resulting tuple $(com_{\mathbf{0}},\ \sigma_{fp})$ is then stored in $f_{\text{FP}}(\texttt{task},\ \ell)$ and used whenever $\mathcal{F}_{\text{PUBA}}$ leaks ℓ during UReg, Bookkeeping or Analytics.

H_{13} Replaces all honest users U by dummy parties, which, when receiving input by $\mathcal{Z}$, forward the input immediately to $\mathcal{F}_{\text{PUBA}}$. During the tasks for UReg, Bookkeeping and Analytics, the new user machines U also leak the input $\texttt{in}_U$ to $\mathcal{S}_{13}$. All messages honest user U sent in H_{12} are now created by $\mathcal{S}_{13}$ in H_{13} and send in the name of U.

Note that all leaks by $\mathcal{F}_{\text{PUBA}}$ are ignored by $\mathcal{S}_{13}$.

H_{14} Replaces all proxies P with dummy parties, which, when receiving input by $\mathcal{Z}$, forward the input directly to $\mathcal{F}_{\text{PUBA}}$. All messages that were sent by honest proxies P in H_{13} are now created by $\mathcal{S}_{14}$ in H_{14} and send in the name of P.

Note that all leaks by $\mathcal{F}_{\text{PUBA}}$ are ignored by $\mathcal{S}_{14}$.

H_{15} $\mathcal{S}_{15}$ now calls $\mathcal{F}_{\text{PUBA}}$ in the name of the corrupted parties with the correct input. This causes $\mathcal{F}_{\text{PUBA}}$ to fully perform as defined by its specification, as all inputs are provided. Hence, instead of computing the function Δ on the inputs in order to simulate $\mathcal{F}_{\text{PPA}}$, $\mathcal{S}_{15}$ now relies on the leaks provided by $\mathcal{F}_{\text{PUBA}}$.

This game also introduces a map f_{LN}, that maps leaked *ssid* values to linking numbers *lin*.

$\mathcal{S}_{15}$ obtains the input of the corrupted parties as follows:

UReg, U corrupted, O honest. After simulating $\mathcal{F}_{\text{PPA}}$, $\mathcal{S}_{15}$ has obtained $\texttt{in}_U$ from the corrupted U. Hence, $\mathcal{S}_{15}$ calls $\mathcal{F}_{\text{PUBA}}$ in the name of the U belonging to pid_U with input $(\texttt{ureg}, \texttt{in}_U)$. Since $\mathcal{F}_{\text{PUBA}}$ now has full input, $\mathcal{S}_{15}$ uses the output $(\alpha, \mathbf{s}, \mathbf{a}, \texttt{out}_U)$ from $\mathcal{F}_{\text{PUBA}}$ to U in order to simulate $\mathcal{F}_{\text{PPA}}$.

Bookkeeping, U corrupted, O honest. After $\mathcal{Z}$ gave instructions to send $(\texttt{bookkeep}(k), \mathcal{UH}, \overline{decom_{\mathcal{UH}}}, com_{fp}, \texttt{in}_U)$ to $\mathcal{F}_{\text{PPA}}$ in the name of U, $\mathcal{S}_{15}$ uses the extracted pid_U (see H_7) to obtain the user U_{pid_U} who registered for the used public key. After verifying that the commitments are valid, that is, $\text{COM.open}(\overline{com_{\mathcal{UH}}}, \overline{decom_{\mathcal{UH}}}, \mathcal{UH}) = 1$ and $\text{COM.open}(com_{fp}, decom_{\mathbf{0}}, \mathbf{0}) = 1$ (see hybrid H_{12}), $\mathcal{S}_{15}$ calls $\mathcal{F}_{\text{PUBA}}$ in the name of U_{pid_U} with input $(\texttt{bookkeep}(k), \texttt{in}_U)$ where $\texttt{in}_U$ has been learned from simulation of $\mathcal{F}_{\text{PPA}}$.

Also, since $\mathcal{F}_{\text{PUBA}}$ now has complete input, $\mathcal{S}_{15}$ obtains leaks. Hence, instead of computing Δ internally, $\mathcal{S}_{15}$ uses the output $(\alpha, \mathbf{s}, \mathbf{a}, \texttt{out}_U)$ from $\mathcal{F}_{\text{PUBA}}$ to U in order generate the output of $\mathcal{F}_{\text{PPA}}$ to U by first computing valid commitment information $(com_{\mathbf{a}}, decom_{\mathbf{a}}) \leftarrow \text{COM.commit}(\mathbf{a})$ and then reporting message $(\alpha, \mathbf{s}, \mathbf{a}, com_{\mathbf{a}}, decom_{\mathbf{a}}, \texttt{out}_U)$.

Outsource, U corrupted, O honest, P honest. $\mathcal{S}_{15}$ obtains the input of the user by using Π_{Share}-*Combine* on the shares U sent to both O and P: after $\mathcal{Z}$ sent instructions to send $(\langle \mathcal{UH} \rangle^O, \langle \texttt{in}_U \rangle^O, \langle \text{o}_{\text{OUT}} \rangle^O, \langle \text{o}_{\text{UNV}} \rangle^O, \langle \text{o}_\alpha \rangle^O, \langle \text{o}_{\mathbf{s}} \rangle^O, \langle \text{o}_{\mathbf{a}} \rangle^O, ser, \overline{com_{id}}, \Pi, com^{(U)}_{ser^{\text{new}}})$ from U to O and to send $(\langle \mathcal{UH} \rangle^P, \langle \texttt{in}_U \rangle^P, \langle \text{o}_{\text{OUT}} \rangle^P, \langle \text{o}_{\text{UNV}} \rangle^P, \langle \text{o}_\alpha \rangle^P, \langle \text{o}_{\mathbf{s}} \rangle^P, \langle \text{o}_{\mathbf{a}} \rangle^P)$ from U to P, $\mathcal{S}_{15}$ uses the shares from both O and P to reconstruct $\texttt{in}_U \coloneqq \Pi_{\text{Share}}\text{-}Combine(\langle \texttt{in}_U \rangle^P, \langle \texttt{in}_U \rangle^O)$; if this fails, $\mathcal{S}_{15}$ sets $\texttt{in}_U \coloneqq \perp$. $\mathcal{S}_{15}$ calls $\mathcal{F}_{\text{PUBA}}$ in the name of U_{pid_U}, where pid_U corresponds to the user whose identity was extracted from Π, using input $(\texttt{outsource}k, \texttt{in}_U)$.

Outsource, U honest, O honest, P corrupted. P is designed to neither learn secrets, nor to have secrets itself. Simulator $\mathcal{S}_{15}$ calls $\mathcal{F}_{\text{PUBA}}$ in the name of P with input $(\texttt{outsource}k)$. Also, $\mathcal{S}_{15}$ sets $f_{\text{ID}}(ssid) \coloneqq lin^{\text{new}}$ after negotiating the linking number.

Outsource, U corrupted, P corrupted, O honest. $\mathcal{S}_{15}$ calls $\mathcal{F}_{\text{PUBA}}$ for both the proxy P (who has no secret input whatsoever) and the user U_{pid_U} (who was identified using the pk_U in Π), but using $\texttt{in}_U \coloneqq \bot$.

Analytics, P corrupted, O honest. Since H_{11}, $\mathcal{S}_{15}$ maintains the list $f_{\text{UI}}^{(O)}$ in the same way an honest operator would. As $\mathcal{S}_{15}$ follows the operators code (due to H_{11}) it stores all the tuples (lin^{new}, $\langle \mathcal{UH} \rangle^O$, $\langle \texttt{in}_U \rangle^O$, $\langle \text{o}_{\text{OUT}} \rangle^O$, $\langle \text{o}_{\text{UNV}} \rangle^O$, $\langle \text{o}_\alpha \rangle^O$, $\langle \text{o}_\mathbf{s} \rangle^O$, $\langle \text{o}_\mathbf{a} \rangle^O$) during the Outsource task; given the additional inputs from simulation of $\mathcal{F}_{\text{PPA}}$ $\mathcal{S}_{15}$ now has a complete view of the used shares.

After receiving instructions from $\mathcal{Z}$ to send a message ($\texttt{analyt}k$, com_{fp}, $\{\langle \mathcal{UH} \rangle^P, \langle \texttt{in}_U \rangle^P, \langle \text{o}_{\text{OUT}} \rangle^P, \langle \text{o}_{\text{UNV}} \rangle^P, \langle \alpha \rangle^P, \langle \mathbf{s} \rangle^P, \langle \mathbf{a} \rangle^P\}_{z=1}^{Z}$) from P to $\mathcal{F}_{\text{PPA}}$, $\mathcal{S}_{15}$ restores each input using the information stored in $f_{\text{UI}}^{(O)}$ and aborts if the reconstruction via Π_{Share}-*Combine* fails. When asked by $\mathcal{F}_{\text{PUBA}}$ for inputs for an index set $I \subset [Z_k]$ of corrupted users $\{U_z\}_{z \in I}$, $\mathcal{S}_{15}$ enters the inputs $\texttt{in}_U$ from the respective indices.

Reconstruction of the output now only happens for parties $z \in I$ in the corrupted party set; information related to honest parties (ct_{α_z}, $ct_{\mathbf{s}_z}$, $ct_{\mathbf{a}_z}$, $ct_{decom_{\mathbf{a}_z}}$, $ct_{\texttt{out}_{U_z}}$) is drawn at random and put to $f_{\text{UI}}^{(P)}$.

Update, any corrupted party. The inputs to the Update-task contain no secrets; so $\mathcal{S}_{15}$ can call $\mathcal{F}_{\text{PUBA}}$ in the name of any corrupted party with input ($\texttt{update}$). For corrupted users, $\mathcal{S}_{15}$ awaits the proof Π to extract the correct user. For corrupted Proxies, $\mathcal{S}_{15}$ awaits their first message. Also, if *only* the proxy is corrupted, $\mathcal{S}_{15}$ awaits the leaked subsession id *ssid* from $\mathcal{F}_{\text{PUBA}}$ to obtain $lin \coloneqq f_{\text{LN}}(ssid)$ and to report the first message from U to P.

H_{16} Introduces a map $f_{\text{OI}}^{(O)}$ and for each proxy a map $f_{\text{OI}}^{(P)}$ to be used by the simulator $\mathcal{S}_{16}$: $f_{\text{OI}}^{(O)}$ is similar to what an honest operator would store for the outsourced information after the Outsource-task, $f_{\text{OI}}^{(P)}$ is the equivalent for the respective proxy.

The map is used during the tasks for Outsource and Analytics:

Outsource, U honest, P honest, O honest. $\mathcal{S}_{16}$ adds a vector of empty entries ($\bot$, $\ldots$, $\bot$) to both $f_{\text{OI}}^{(P)}(k)$ and $f_{\text{OI}}^{(O)}(k)$.

Outsource, U corrupted, P honest, O honest. lin^{new} is now randomly sampled by $\mathcal{S}_{16}$, which replaces the coin-toss of O and P.

After receiving instructions from $\mathcal{Z}$ to send a message ($\langle \mathcal{UH} \rangle^O$, $\langle \texttt{in}_U \rangle^O$, $\langle \text{o}_{\text{OUT}} \rangle^O$, $\langle \text{o}_{\text{UNV}} \rangle^O$, $\langle \text{o}_\alpha \rangle^O$, $\langle \text{o}_\mathbf{s} \rangle^O$, $\langle \text{o}_\mathbf{a} \rangle^O$, ser, $\overline{com_{id}}$, Π, $com_{ser^{\text{new}}}^{(U)}$) from a corrupted U to O, $\mathcal{S}_{16}$ adds a new entry (lin^{new}, $\langle \mathcal{UH} \rangle^O$, $\langle \texttt{in}_U \rangle^O$, $\langle \text{o}_{\text{OUT}} \rangle^O$, $\langle \text{o}_{\text{UNV}} \rangle^O$, $\langle \text{o}_\alpha \rangle^O$, $\langle \text{o}_\mathbf{s} \rangle^O$, $\langle \text{o}_\mathbf{a} \rangle^O$) to $f_{\text{OI}}^{(O)}$.

After receiving instructions from $\mathcal{Z}$ to send a message ($\langle \mathcal{UH} \rangle^P$, $\langle \texttt{in}_U \rangle^P$, $\langle \text{o}_{\text{OUT}} \rangle^P$, $\langle \text{o}_{\text{UNV}} \rangle^P$, $\langle \text{o}_\alpha \rangle^P$, $\langle \text{o}_\mathbf{s} \rangle^P$, $\langle \text{o}_\mathbf{a} \rangle^P$) from a corrupted U to P, $\mathcal{S}_{16}$ adds a

new entry $(lin^{\text{new}}, \langle \mathcal{UH} \rangle^P, \langle \mathtt{in}_U \rangle^P, \langle \mathtt{o}_{\text{OUT}} \rangle^P, \langle \mathtt{o}_{\text{UNV}} \rangle^P, \langle \mathtt{o}_\alpha \rangle^P, \langle \mathtt{o}_\mathbf{s} \rangle^P, \langle \mathtt{o}_\mathbf{a} \rangle^P)$ to $f_{\text{OI}}^{(P)}$.

Outsource, U honest, P corrupted, O honest. When the honest U is supposed to create shares of $\mathcal{UH}$, $\mathtt{in}_U$, $\mathtt{o}_{\text{OUT}}$, $\mathtt{o}_{\text{UNV}}$, $\mathtt{o}_\alpha$, $\mathtt{o}_\mathbf{s}$ and $\mathtt{o}_\mathbf{a}$, $\mathcal{S}_{16}$ creates shares of the zero-vector using Π_{Share}-$Share(\mathbf{0})$ and uses those in the same way the user uses the actual shares in the protocol. After having the linking number lin^{new} created honestly, $\mathcal{S}_{16}$ stores the values in $f_{\text{OI}}^{(O)}$ and ignores $f_{\text{OI}}^{(P)}$.

Outsource, U corrupted, P corrupted, O honest. $\mathcal{S}_{16}$ follows the protocol of O regarding f_{OI}, that is, after $\mathcal{Z}$ sent instructions to send a message $(\langle \mathcal{UH} \rangle^O, \langle \mathtt{in}_U \rangle^O, \langle \mathtt{o}_{\text{OUT}} \rangle^O, \langle \mathtt{o}_{\text{UNV}} \rangle^O, \langle \mathtt{o}_\alpha \rangle^O, \langle \mathtt{o}_\mathbf{s} \rangle^O, \langle \mathtt{o}_\mathbf{a} \rangle^O, ser, \overline{com_{id}}, \Pi, com_{ser^{\text{new}}}^{(U)})$ from U to O and after $\mathcal{S}_{16}$ took the role of O in honestly computing the linking number, it adds $(lin^{\text{new}}, \langle \mathcal{UH} \rangle^O, \langle \mathtt{in}_U \rangle^O, \langle \mathtt{o}_{\text{OUT}} \rangle^O, \langle \mathtt{o}_{\text{UNV}} \rangle^O, \langle \mathtt{o}_\alpha \rangle^O, \langle \mathtt{o}_\mathbf{s} \rangle^O, \langle \mathbf{a} \rangle^O)$ to $f_{\text{OI}}^{(O)}$.

H_{17} All remaining honest parties are replaced by dummy parties, which, upon receiving input by $\mathcal{Z}$, only forward their input into $\mathcal{F}_{\text{PUBA}}$.

H_{18} All messages between honest parties are simulated by having $\mathcal{S}_{18}$ report messages of zero-vectors of correct size. Consequently, all operations that do not result in messages are removed.

H_{19} During UReg with an honest user, instead of honestly creating a user id id, interpreting it as secret key and computing a public key from it, $\mathcal{S}_{19}$ uniformly samples a public key $\mathsf{pk}_U \xleftarrow{\text{r}} G_1$ directly.

H_{20} If P is corrupted and U and O are honest, $\mathcal{S}_{20}$ changes its behavior during simulation of the Outsource-task. Instead of calling Π_{Verify} on the whole logbook, $\mathcal{S}_{20}$ only verifies that the linking number lin^{new} received by P confirms with the two shares $(lin^{\text{new}})^{(O)} + (lin^{\text{new}})^{(P)}$.

We now prove lemmata claiming indistinguishability of each consecutive pair of games against every PPT-environment $\mathcal{Z}$. Thus, with H_1 being the real world and H_{20} corresponding to the ideal world with a simulator acting as described above, we have proven the protocol Π_{PUBA} to be as secure as the functionality $\mathcal{F}_{\text{PUBA}}$.

Lemma A.3.23 (Indistinguishability of H_1 and H_2)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_1 and H_2 with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. $\mathcal{F}_{\text{PPA}}$, $\mathcal{F}_{\text{KE}}$ and $\mathcal{F}_{\text{BB}}$ are considered to be *hybrid functionalities*, which can be instantiated by a real, UC-secure protocol. UC-security of the instantiations implies, that $\mathcal{F}_{\text{BB}}$, $\mathcal{F}_{\text{KE}}$ and $\mathcal{F}_{\text{PPA}}$ all have a *simulator*, which runs in polynomial time and which can provide the view of a real protocol execution to any PPT-environment $\mathcal{Z}$, but is only using the ideal functionalities, for any given corruption scenario. Hence, $\mathcal{S}_2$ can execute this code without runtime restrictions, whenever an interaction between any party and either $\mathcal{F}_{\text{BB}}$, $\mathcal{F}_{\text{KE}}$ or $\mathcal{F}_{\text{PPA}}$ is requested by $\mathcal{Z}$.

Indistinguishability now easily follows from the security of $\mathcal{F}_{\text{BB}}$, $\mathcal{F}_{\text{KE}}$ and $\mathcal{F}_{\text{PPA}}$. Their UC-security implies, that the resp. protocol (which is used in H_1) cannot be distinguished from the simulated view (which is used in H_2). If $\mathcal{Z}$ could differentiate H_1 and H_2 with non-negligible advantage over guessing, we can build a distinguisher $\mathcal{Z}'$ for the real and ideal view of the resp. hybrid functionalities as follows:

- $\mathcal{Z}'$ internally simulates all users U, all proxies P, the operator O and the environment $\mathcal{Z}$ that can distinguish H_1 and H_2 with probability $\frac{1}{2} + \mathcal{A}$ by executing their code in its head.
- $\mathcal{Z}'$ uses the distinguishing algorithm to let $\mathcal{Z}$ output a single bit: 0 for H_1 and 1 for H_2.
- $\mathcal{Z}'$ outputs this bit, where 0 means that it is in the real world and 1 means it is in the ideal world.

Note that the environment $\mathcal{Z}'$ also has a distinguishing advantage of $\frac{1}{2} + \mathcal{A}$. UC-security of $\mathcal{F}_{\text{BB}}$, $\mathcal{F}_{\text{KE}}$ and $\mathcal{F}_{\text{PPA}}$ thus require $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$, which concludes the proof. □

Lemma A.3.24 (Indistinguishability of H_2 and H_3)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_2 and H_3 with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. The simulator mimics the behavior of an honest operator. Since none of the code regarding L_{SER} is in any way dependent on the secret input $\mathcal{Z}$ gives to U or O, this can be done without any loss of generality. In particular, assume that any party (that is, either O or $\mathcal{S}_3$) aborts due to a duplicate serial number in any game. In either case, this would mean that there was a previous interaction of either the task for Bookkeeping, Outsource, or Update, where U opened a value of com_{ser} to the same *ser* that is now seen by the respective party. Since the different parties execute the same code, their abort-criteria is equivalent. The same is true for the verification of the proof Π. □

Lemma A.3.25 (Indistinguishability of H_3 and H_4)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_3 and H_4 with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Indistinguishability here holds for the same reason that it held in Lemma A.3.24. The contents of $f_{\text{UI}}^{(P)}$ and $f_{\text{UI}}^{(O)}$ only depend on *messages* which $\mathcal{S}_4$ can access via simulation of $\mathcal{F}_{\text{PPA}}$, and not (directly) on *secret input*, which is hidden from it. Hence, we only have a new encapsulation, where (PPT-)code that depends only on previous messages was executed by O or P in H_3 is now executed by $\mathcal{S}_4$ in H_4.

The equivalence of their contents easily follows from the same argument. The contents themselves depend on the message that U sent to the resp. parties, the order depends on the (adversarialy chosen) scheduling mechanism, which, given any environment $\mathcal{Z}$ that tries to distinguish the two games, is equivalent.

The final change contains the check in the corruption scenario where P is corrupted, but U and O are honest. Here, in H_3, U aborts if either $\mathsf{open}(com_{\mathbf{a}}, decom_{\mathbf{a}}, \mathbf{a}) \neq 1$, or if ct_{out_U} differs from the value the honest receiver sent.

Equivalence for the latter is straightforward. If U aborts due to the former condition ($\mathsf{open}(com_{\mathbf{a}}, decom_{\mathbf{a}}, \mathbf{a}) \neq 1$), then this means that some value was tampered with. $com_{\mathbf{a}}$ was received by O, who, in this scenario, is honest, hence it is correct. The one-time pads $o_{\mathbf{a}}$, o_{UNV} used to mask $\mathbf{a}$ and $decom_{\mathbf{a}}$ were created by the user U during the Outsource task and hence are also correct. Thus, the only values that *could* be tampered with are $ct_{\mathbf{a}}$ and $ct_{decom_{\mathbf{a}}}$, which $\mathcal{S}_4$ has seen during the Outsource task and hence, for which a consistency check suffices. □

Lemma A.3.26 (Indistinguishability of H_4 and H_5)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_4 and H_5 with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Indistinguishability here follows from the trapdoor nature of POK. If any environment $\mathcal{Z}$ could distinguish the execution of the protocol when using *crs* created by $crs \leftarrow \mathsf{SetupPoK}$ from *crs* created by $(crs, td_{\mathrm{ext}}) \leftarrow \mathsf{SetupEPoK}$ with probability $\frac{1}{2} + \mathcal{A}$, we can build a PPT-environment $\mathcal{Z}'$ that breaks the indistinguishability of the dual-mode property of POK, by having $\mathcal{Z}'$ execute the code of all parties in its head. This leads to the same success probability of $\frac{1}{2} + \mathcal{A}$, thus causing $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$ by requirement of the chosen POK-scheme. □

Lemma A.3.27 (Indistinguishability of H_5 and H_6)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_5 and H_6 with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. The only difference between H_5 and H_6 is, that $\mathcal{S}_6$ stores additional information in H_6 that was accessible even in H_5. Additionally, this game introduces a new abort-criteria.

For indistinguishability of H_5 and H_6, we thus have to show that those two criteria are, in fact, equivalent. In H_5, O fetches the key pk_U belonging to pid_U from $\mathcal{F}_{\mathrm{BB}}$, thus effectively asking (since H_2) $\mathcal{S}_6$ for the key that pid_U registered there. Thus, instead of $\mathcal{S}_6$ simulating $\mathcal{F}_{\mathrm{BB}}$ and giving O the key pk_U so O can verify that pk_U is the one that was used in Π, $\mathcal{S}_6$ now does the exact same thing, only that, due to the simulation of $\mathcal{F}_{\mathrm{BB}}$, no further interaction is required to obtain pk_U. Hence, aborts in H_5 due to a duplicate or mismatching pk_U occur if and only if aborts in H_6 happen due to a duplicate or mismatching pk_U. Thus, both distributions from H_5 and H_6 are equivalent. □

Lemma A.3.28 (Indistinguishability of H_6 and H_7)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_6 and H_7 with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Extraction is possible due to the different *crs* introduced in H_5 and the fact that each witness only has elements from the target group, so F-extractability of POK suffices to obtain pk_U. The public key pk_U is contained in every language used throughout the system. The user found by looking up $pid_U \coloneqq f_{\mathrm{ID}}(\mathsf{pk}_U)$ is the correct user.

Assume for the sake of contradiction that $\mathcal{S}_7$ aborts in H_7 because $pid_U = \bot$, but $\mathcal{S}_6$ would successfully terminate in H_6. The following behavior by U could have caused this:

U **proved a faulty statement.** The correctness property of POK would cause this to be detected in H_6 by O with overwhelming probability, thus causing an abort.

U **changed the commitment** com_{id} **in the statement.** Then the proof of equivalence between com_{id} and the rerandomized $\overline{com_{id}}$ would have to be forged, thus breaking the statement of Π. This would also cause an abort in H_7, due to the correctness of POK.

U **opened** com_{id} **to a different** pk'_U**.** In this case, the binding-property of the commitment scheme COM would be violated. Since we assumed COM to be unconditionally binding, this cannot occur.

U **created its own signature.** In case the user U created a signature σ on a new logbook Log' containing a public key $\mathsf{pk}'_U \neq \mathsf{pk}_U$ without knowing the operator's secret key sk_O, U would break the unforgeability of SIG. By requirement on SIG, this is possible only with probability negligible in λ.

The reverse is also true; if O discards the proof Π in H_6, then they do so too in H_7, as this part has not been changed. □

Lemma A.3.29 (Indistinguishability of H_7 and H_8)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_7 and H_8 with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Indistinguishability of those two games trivially follows; the new machine is only accessible by *honest* parties, who follow the protocol. Their protocol description in H_8 does not include any access to $\mathcal{F}_{\text{PUBA}}$. The simulator doesn't access $\mathcal{F}_{\text{PUBA}}$ either, so the two distributions of H_7 and H_8 are statistically close and hence, the best any PPT-environment $\mathcal{Z}$ can do is guessing. □

Lemma A.3.30 (Indistinguishability of H_8 and H_9)**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_8 and H_9 with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Indistinguishability of the two games directly follows from the facts that (1) both tasks for Init and SignFP that contain the TSA $\mathcal{TSA}$ are executed with an honest operator O, and (2) none of the two aforementioned tasks contain any secret inputs of $\mathcal{TSA}$. Hence, the role of $\mathcal{TSA}$ can be played by $\mathcal{S}_9$, who creates the same message distribution. This implies indistinguishability based on the fact that in the view of $\mathcal{Z}$, the two games are equivalent. □

Lemma A.3.31 (Indistinguishability of H_9 and H_{10})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_9 and H_{10} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. The rewriting is purely cosmetical, as essentially the same code is executed on different machines. In H_9, the commitment com_{fp} is computed by O, the signature σ_{fp} is computed by $\mathcal{TSA}$, and the resp. values are fetched by O prior to a computation with $\mathcal{F}_{\text{PPA}}$. In H_{10}, all these steps are done by the simulator. Since $\mathcal{TSA}$ is honest and its key has thus been created by $\mathcal{S}_{10}$ and the relevant information for com_{fp} and σ_{fp} are leaked by O, the honest code can be executed directly.

Finally, the last change induced is the replaced check with the decommitment information. Since this is essentially a rewriting, this change is purely cosmetical and hence undetectable. □

Lemma A.3.32 (Indistinguishability of H_{10} and H_{11})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{10} and H_{11} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. O is a PPT-machine, which executes code based on the secret input from $\mathcal{Z}$. Knowing this input from the leak, $\mathcal{S}_{11}$ can execute the code of the honest O by following the protocol. This trivially leads to statistical indistinguishability. □

Lemma A.3.33 (Indistinguishability of H_{11} and H_{12})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{11} and H_{12} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. The major change induced in this game hop has the simulator reporting valid zero-commit-ments instead of valid commitments on function parameters fp. Let $\mathcal{Z}$ be an environment that distinguishes the two games from H_{11} and H_{12}. We construct an adversary $\mathcal{A}$ that breaks the hiding property of COM, which we model similar to IND-CPA for encryption schemes. We adapt the LR-view, stating that the challenger C on the hiding game provides us with an oracle that accepts two different inputs, but outputs a valid commitment on a fixed one of them.

The adversary $\mathcal{A}$ can thus create the transcript from H_{11}, but whenever a commitment com_{fp} on the FPs fp is required, $\mathcal{A}$ sends the two messages $(\mathbf{0},\ fp)$ to C and obtains a commitment on one of them.

Note that if the commitment always uses the former entry, $\mathcal{A}$ perfectly simulates H_{12}, and if the commitment always uses the latter entry, $\mathcal{A}$ perfectly simulates H_{11}.

It thus follows that the success probability of $\mathcal{Z}$ in detecting this game hop is limited by the probability of $\mathcal{A}$ to break the hiding game, which is negligible. □

Lemma A.3.34 (Indistinguishability of H_{12} and H_{13})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{12} and H_{13} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. The case here is similar to that from Lemma A.3.32. $\mathcal{S}_{13}$ can execute the code of any honest user U, since honest user reveal their identity to $\mathcal{S}_{13}$; by leaking the secret input, $\mathcal{S}_{13}$ can follow the protocol of U from H_{12}. Indistinguishability follows. □

Lemma A.3.35 (Indistinguishability of H_{13} and H_{14})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{13} and H_{14} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. The proxy itself has no secrets, so no leaks are required here. Hence, all messages of P depend only on messages it has seen before. Since $\mathcal{S}_{14}$ can see those messages as well and P is a PPT-machine, $\mathcal{S}_{14}$ can execute the code of honest proxies, thus causing a statistically indistinguishable distribution from H_{13}. □

Lemma A.3.36 (Indistinguishability of H_{14} and H_{15})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{14} and H_{15} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Indistinguishability of H_{14} and H_{15} follows from the fact that in both cases the messages depend on exactly the same values. Essentially, the same code is executed, only by different machines; the game hop is only a cosmetical one.

In more detail, the situation looks as follows:

UReg, U corrupted, O honest. Here, the definition of $\mathcal{F}_{\text{PUBA}}$ and $\mathcal{S}_{15}$, who calls $\mathcal{F}_{PUBA}^{(\Delta)}$-*UReg* with the correct inputs $\texttt{in}_U$, implies correct behavior. The input $\texttt{in}_U$ is correct due to the extractability and the pid is correct as a wrong pid would require the user to break the Co-CDH assumption by computing the id *id* of the public key pk_U stored in $\mathcal{F}_{\text{BB}}$.

Bookkeeping, U corrupted, O honest. In H_{14}, the user holds its own logbook Log with the corresponding user history $\mathcal{UH}$. In H_{15}, $\mathcal{F}_{\text{PUBA}}$ executes the same function Δ on the inputs to $\mathcal{F}_{\text{PUBA}}$, that the simulator computed in H_{14}. In H_{15}, the correct – and latest – input is used by definition of the ideal functionality. The transfer values $(\alpha, \mathbf{s}, \mathbf{a})$ are output to U and hence visible to $\mathcal{S}_{15}$ and the commitment and decommitment on $\mathbf{a}$ can be computed directly by $\mathcal{S}_{15}$. If this value would have been the same in H_{14} then indistinguishability for any exectution of the Bookkeeping-task trivially follows; the best any *PPT*-environment $\mathcal{Z}$ could do here is to guess.

So assume that there is some set of inputs, for which $\mathcal{Z}$ can differentiate between H_{14} and H_{15} notably better than guessing based on the Bookkeeping task. Since the computation performed by the simulator in H_{14} is exactly the same as the one $\mathcal{F}_{\text{PUBA}}$ does in H_{15} and $\mathcal{Z}$ cannot lie about $\texttt{in}_U$ (as it is input into $\mathcal{F}_{\text{PPA}}$), the only way $\mathcal{Z}$ could try to win here is by providing different input $\mathcal{UH}$. There are different ways $\mathcal{Z}$ could achieve that:

- $\mathcal{Z}$ provides a wrong proof regarding σ. Then, we could build an environment that either forges a signature σ and uses an honest witness, or fakes a proof Π_{Val} and uses a false witness. The former would contradict our EUF-CMA requirement for SIG, the latter would contradict the soundness-property required for POK.

- $\mathcal{Z}$ provides the correct information of a different corrupted user U'. We assumed SIG to be EUF-CMA-secure, thus assuring unforgeability of the signature σ on Log. Hence, $\mathcal{S}_{15}$ would extract the id pk_U of U'. By a lookup from f_{ID}, $\mathcal{S}_{15}$ would get the *pid* of U' and provide input to $\mathcal{F}_{\text{PUBA}}$ in the name of U'. Thus, $\mathcal{F}_{\text{PUBA}}$ uses the same $\mathcal{UH}$, that would have been used by the simulator in H_{14}.

Hence, assuming that $\mathcal{F}_{\text{PUBA}}$ internally updates $\mathcal{UH}$ correctly (which we will show for the other tasks as well), this change cannot be used to increase the chance of $\mathcal{Z}$ to differentiate.

Outsource, U corrupted, O honest, P honest. Here, too, indistinguishability trivially follows from the correctness of inputs; we merely copied the reconstruction of shares for this scenario from Analytics to Outsource. Those are input directly into $\mathcal{F}_{\text{PUBA}}$, where they are used later. The shares $\langle \mathtt{in}_U \rangle^P$ and $\langle \mathtt{in}_U \rangle^O$ are either correct, in which case $\mathcal{F}_{\text{PUBA}}$ will use them accordingly during Analytics. Or they are not, in which case $\mathcal{S}_{15}$ sets $\mathtt{in}_U \coloneqq \bot$; the reconstruction H_{14} would fail during the Outsource-task, which also happens in H_{15}. With the two parties performing Analytics being honest, no further problems arise during the subsequent execution. The extraction of pk_U further removes the ability of $\mathcal{Z}$ to cheat by letting $\mathcal{S}_{15}$ send input to $\mathcal{F}_{\text{PUBA}}$ in the name of the wrong U.

Outsource, U honest, P corrupted, O honest. In this case, it is not possible to cheat without being detected. By knowing the input $\mathcal{Z}$ would have given to P, $\mathcal{S}_{15}$ can mimic the behavior of an honest dummy proxy and forward it into $\mathcal{F}_{\text{PUBA}}$.

Outsource, U corrupted, P corrupted, O honest. The correct user U can be determined via extraction of Π, so no $\mathcal{Z}$ cannot use two different users for the two games here. However, in this task, $\mathcal{S}_{15}$ information-theoretically cannot determine the correct input $\mathtt{in}_U$, as it only sees one part $\langle \mathtt{in}_U \rangle^P$ of the additive sharing—the second part, $\langle \mathtt{in}_U \rangle^O$, is sent between two corrupted parties and hence not visible for $\mathcal{S}_{15}$. However, in the subsequent Analytics execution, $\mathcal{S}_{15}$ learns the shares that U sent to P via simulation of $\mathcal{F}_{\text{PPA}}$. There, $\mathcal{S}_{15}$ can reconstruct $\mathtt{in}_U$. Note that there is no difference between $\mathcal{S}_{15}$ learning the input during the Outsource task and inserting it into $\mathcal{F}_{\text{PUBA}}$ right away and $\mathcal{S}_{15}$ learning it during the Analytics task, since $\mathcal{F}_{\text{PUBA}}$ allows $\mathcal{S}_{15}$ to update the input $\mathtt{in}_U$ for all corrupted users before starting the computation during Analytics. Thus, the same input $\mathtt{in}_U$ is used in both games.

Analytics, P corrupted, O honest. Here, $\mathcal{S}_{15}$ still uses the same shares for the operator in both games and receives (and verifies) all proxy-shares input into $\mathcal{F}_{\text{PPA}}$ in both games, there is no direct change here. The only new thing is the equivocation of $\mathtt{in}_U$ for corrupted users. This change was already discussed above; the remaining protocol remains equivalent, since $\mathcal{S}_{15}$ only does in H_{15} what honest parties would do in H_{14}.

Update, U corrupted, P honest, O honest. The values that determine the messages are still equivalent in both games. The values for $f_{\text{UI}}^{(P)}$ and $f_{\text{UI}}^{(O)}$ were honestly kept by

$\mathcal{S}_{15}$, so both just follow the same protocol. As nothing is done with the output of $\mathcal{F}_{\text{PUBA}}$, distinguishing here is not possible.

Update, U honest, P corrupted, O honest. The linking number that was sent as leak by U to the simulator in H_{14} trivially equals $f_{\text{LN}}(\mathit{ssid})$ that was kept by $\mathcal{S}_{15}$ in H_{15}. During the Outsource task, assuming lin is sampled from a sufficiently large space, then the probability that the mapping $\mathit{lin} \rightarrow \mathit{ssid}$ is unique becomes overwhelming. During the Update task, this does not change. If this wouldn't be the case, $\mathcal{Z}$ would have to create a duplicate lin, which, with Blum coin toss, is possible only with negligible probability. Even then, $\mathcal{S}_{15}$ in H_{15} would use the correct linking number lin. Note further that $\mathcal{Z}$ has no way on how to lie about ssid.

Update, U corrupted, P corrupted, O honest. Here, the simulator only has to ensure that the interaction between O and U is canonical. Again, nothing here depends on the output of $\mathcal{F}_{\text{PUBA}}$, it is only called to keep the functionality in a consistent space. Hence, the changes induced here provide no advantage to $\mathcal{Z}$ in distinguishing the two games.

□

Lemma A.3.37 (Indistinguishability of H_{15} and H_{16})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{15} and H_{16} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. We claim indistinguishability based on the fact that the two distributions from H_{15} and H_{16} are indistinguishable.

First, notice that the binary outcome of a Blum coin toss and a uniformly random bit cannot be differentiated better that by guessing, so this change doesn't provide any distinguishing advantage.

To support our claim also regarding the new map f_{UI} we consider all possible corruption cases:

P honest, O honest. In case U is also honest, the functionality $\mathcal{F}_{\text{PUBA}}$ directly obtains input from the respective parties during the Outsource task, which causes $\mathcal{F}_{\text{PUBA}}$ to load the correct $\mathcal{UH}$ and to use the input $\mathtt{in}_U$ provided by U. Hence, the simulator has to only remember *that* honest parties provided input, not *what* these inputs were. Thus, inserting the special symbol $\perp$ to keep the list size consistent suffices, since those values are never used again.

In case U is corrupted, $\mathcal{Z}$ has to send shares $\langle(\cdot)\rangle^P$ and $\langle(\cdot)\rangle^O$ in the name of U to P and O, respectively. Neither of them are corrupted, so $\mathcal{S}_{16}$ can see both messages. Since neither O nor P have secret inputs, $\mathcal{S}_{16}$ can follow the honest protocols, thus producing the same distribution.

P **corrupted,** *O* **honest.** If the user U is honest, then $\mathcal{S}_{16}$ has to send messages containing valid shares of Us input, without actually knowing the input. Thus, $\mathcal{S}_{16}$ distributes zero-shares; the environment $\mathcal{Z}$ can see only the part sent to P, not the one sent to O. Obviously, no environment $\mathcal{Z}$ can distinguish its part of the zero-sharing obtained in H_{16} from a valid sharing of the respective user input from H_{15} better than by randomly guessing. Hence, those distributions look equivalent. The simulator follows the protocol of O regarding f_{OI} honestly, which causes no difference in the distributions.

Against a *corrupted* user U, $\mathcal{S}_{16}$ directly follows the protocol of O. This trivially causes the same distribution.

Since all possible cases cause indistinguishable distributions, our claim follows. □

Lemma A.3.38 (Indistinguishability of H_{16} and H_{17})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{16} and H_{17} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Again, indistinguishability follows from the fact that the messages sent are still the same. All the leaks sent by honest parties to the simulator in H_{16} are ignored, the messages are independent of any leaks still sent by the semi-dummy parties: the only change between H_{16} and H_{17} is with respect to $\mathtt{in}_O$ and $\mathtt{in}_U$. Both are input to $\mathcal{F}_{\text{PUBA}}$ directly by the respective dummy party and used there accordingly. Hence, the change induced by H_{17} doesn't change the view of $\mathcal{Z}$ at all, which makes indistinguishability trivial. □

Lemma A.3.39 (Indistinguishability of H_{17} and H_{18})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{17} and H_{18} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. We generally assume that parties obtain pairwise shared keys at the beginning of each task from $\mathcal{F}_{\text{KE}}$. This yields a key k, which they use for symmetric encryption with the IND-CPA-secure symmetric encryption scheme ENC. Assume for the sake of contradiction that there is a PPT environment $\mathcal{Z}$, which can distinguish between H_{17} and H_{18} with advantage $\frac{1}{2} + \mathcal{A}$ for $\mathcal{A} \notin \mathsf{negl}(\lambda)(\lambda)$. We show that this implies an adversary $\mathcal{A}$ on the IND-CPA experiment of ENC:

We adapt the LR-view, that provides the reduction algorithm with an algorithm that on input m_0, m_1 either always outputs $\mathsf{ENC.Encrypt}(\mathsf{sk},\ m_0)$, or $\mathsf{ENC.Encrypt}(\mathsf{sk},\ m_1)$.

$\mathcal{Z}$ distinguishes H_{17} and H_{18}, and $\mathcal{A}$ can simulate the views correctly, by providing always the oracle output on input $(m_0,\ \mathbf{0})$, where m_0 is the honest transcript; If the adversary outputs H_{18}, $\mathcal{A}$ sends 0 to C. If the adversary outputs H_{17}, $\mathcal{A}$ outputs 1 to C.

Note that $\mathcal{A}$ has the same success probability as $\mathcal{Z}$, that is, $\frac{1}{2}+\mathcal{A}$. Hence, by our assumption, it follows that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$, which concludes our proof. □

Lemma A.3.40 (Indistinguishability of H_{18} and H_{19})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{18} and H_{19} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. By assumption on the group gp, a public key is uniformly distributed in G_1. Any environment $\mathcal{Z}$ distinguishing H_{18} and H_{19} based on pk_U would violate this assumption. The simulator now lacks knowledge of the corresponding user id *id*. However, note that *id* is never used in H_{18}, so $\mathcal{S}_{19}$ can create a similar distribution in H_{19} independently of *id*. □

Lemma A.3.41 (Indistinguishability of H_{19} and H_{20})**.** *Let $\mathcal{Z}$ be a PPT-environment. Let $\mathcal{Z}$ distinguish H_{19} and H_{20} with probability $1/2 + \mathcal{A}$. Then it holds that $\mathcal{A} \in \mathsf{negl}(\lambda)(\lambda)$.*

Proof. Indistinguishability easily follows from the fact that in H_{19}, the only corrupted party is P and the only value depending on the proxy is *lin*. Hence, Π_{Verify} only aborts, iff P sent a wrong linking number *lin*. This is still the case in H_{20}, thus making indistinguishability trivial. □

Thus, we can now finally prove our final security statement:

Corollary A.3.42 (System Security)**.** *For all environments $\mathcal{Z}$ who statically corrupted a subset $U' \subseteq U$ and the proxy P, it follows that*

$$\Pi_{\text{PUBA}}{}^{(\mathcal{F}_{\text{PPA}}, \mathcal{F}_{\text{BB}}, \mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{SMT}}^{\text{os-auth}}, \mathcal{F}_{\text{CRS}})} \geq_{UC} \mathcal{F}_{\text{PUBA}}$$

We have shown in Lemma A.3.23 to Lemma A.3.41, that for an honest operator, the simulator $\mathcal{S}_{\text{SysSec}}$ acting in the ideal world can provide a view for $\mathcal{Z}$ that is indistinguishable from a real execution of the protocol:

$$view_{\mathcal{Z},\mathcal{A},\Pi_{\text{PUBA}}} \approx^{c} view_{\mathcal{Z},\mathcal{S}_{\text{SysSec}},\mathcal{F}_{\text{PUBA}}}$$

Thus, by combining Corollaries A.3.22 and A.3.42, our main claim follows:

Corollary A.3.43 (Security)**.** *Assuming that* SIG *is an EUF-CMA-secure structure preserving signature scheme,* POK *is a trapdoor dual-mode NIZKPoK scheme,* $\mathsf{gp} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, p, G_1, G_2)$ *is a pairing-group where the Co-CDH assumption is hard,* COM *is an unconditionally hiding (and computationally binding) homomorphic commitment scheme, where images of* COM *can efficiently be inverted, then for all environments $\mathcal{Z}$ who do not corrupt the operator O* and *the proxy P at the same time, it holds that*

$$\Pi_{\text{PUBA}}{}^{(\mathcal{F}_{\text{PPA}}, \mathcal{F}_{\text{BB}}, \mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{SMT}}^{\text{os-auth}}, \mathcal{F}_{\text{CRS}})} \geq_{UC} \mathcal{F}_{\text{PUBA}}$$

It follows that our protocol Π_{PUBA} is at least as secure as the ideal functionality $\mathcal{F}_{\text{PUBA}}$.

B. Appendix for Chapter 4

This chapter contains some additional information for POBA, mainly the formal proof of security in Section B.2. Additionally, Section B.1 gives more details on the zero-knowledge proofs used in our system, including the concrete pairing-product equations that we prove.

B.1. Implementation Details of the NIZK

To achieve efficient straight-line simulation-extractability for our statements, we augment the Groth–Sahai (GS) non-interactive proof system for pairing product equations [GS08; EG14] with the transformation described in [Gro06]. The GS proof system is a dual-mode system in the common reference string model, which can be setup either in zero-knowledge mode with a simulation trapdoor or in soundness mode with an extraction trapdoor. We achieve straight-line extraction by using the extraction trapdoor, achieve zero-knowledge simulation by standard transformation of generating an OR-proof of either the intended statement or knowledge of a new simulation trapdoor (a signature under a verification key that will also be part of the crs), and finally achieve simulation-extractability (and thus also non-malleability) by the transformation described in [Gro06], i.e., adding a one-time-signature scheme, generating a fresh keypair when generating a new proof, adding the verification key to the statement and then signing the complete statement and proof.

The concrete PPE of the zero-knowledge branch are shown in Fig. B.1a. The concrete pairing product equations (PPE) proved in the Groth–Sahai proof system in our implementation are provided in Figs. B.1b, B.1c and B.2. We use the notation of [EG14], where variables of the statement have type `G1_pub` or `G2_pub` for elements of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, and variables of the statement can be of type either `Gi_com` or `Gi_enc`. Additionally, there are variables `g1` and `g2` of type `G1_base` and `G2_base` representing the generators of $\mathbb{G}_1$ and $\mathbb{G}_2$.

PPE for ZK simulation branch

```
    / Statement
1 : G1_pub zkPK    / simulation pk from crs
2 : G2_pub zkHash    / hash of statement
    / Witness
3 : G2_enc zkSig    / BLS signature on zkHash
    / Equations
4 : e(zkPK,g2) + e(g1,zkHash) - e(g1,zkSig) = 0
```

(a) The concrete equations of the simulation branch.

PPE for $\mathcal{R}_{\mathsf{UR}}$

```
    / Statement
1 : G1_pub pk
    / Witness
2 : G2_enc sk
    / Equations
3 : e(pk,g2) + e(g1,sk) = 0
```

(b) The concrete equations to implement $\mathcal{R}_{\mathsf{UR}}$.

PPE for $\mathcal{R}_{\mathsf{PE}}$

```
    / Statement
1 : G1_pub pk    / pk_U
2 : G1_pub addr    / addr
3 : G1_pub ek    / ek
4 : G1_pub ct1    / ciphertext of addr
5 : G1_pub ct2
    / Witness
6 : G2_enc sk    / sk_U
7 : G2_com rinv    / r^-1
    / Equations
8 : /  encryption of addr
9 : e(ct1,g2) + e(g1,rinv) = 0
10 : e(ct2,g2) + e(ek,rinv) - e(addr,g2) = 0
11 : /  knowledge of sk_U
12 : e(pk,g2) + e(g1,sk) = 0
```

(c) The concrete equations to implement $\mathcal{R}_{\mathsf{PE}}$.

Figure B.1.: The concrete pairing product equations to realize the different zero-knowledge proofs.

PPE for $\mathcal{R}_{\mathsf{CE}}$

	/ Statement		/ Witness
1 :	G1_pub ct1 / ciphertext of *addr*	1 :	G1_com pk
2 :	G1_pub ct2	2 :	G1_com addr
3 :	G1_pub ek / ek	3 :	G2_enc sk
4 :	G2_pub ppA11 / crs of TSIG	4 :	G2_com rinv / r^{-1}
5 :	G2_pub ppA21	5 :	G1_com fsig11 / TSIG signature
6 :	G2_pub ppUA11	6 :	G1_com fsig12
7 :	G2_pub ppUA21	7 :	G1_com ssig11
8 :	G2_pub ppVA11	8 :	G1_com ssig12
9 :	G2_pub ppVA21	9 :	G1_com tsig11
10 :	G1_pub ppB11	10 :	G1_com tsig12
11 :	G1_pub ppB21	11 :	G2_enc fosig
12 :	G1_pub ppBtU11		
13 :	G1_pub ppBtU12		
14 :	G1_pub ppBtV11		
15 :	G1_pub ppBtV12		
16 :	G2_pub vk11		
17 :	G2_pub vk21		
18 :	G2_pub vk31		
19 :	G2_pub h / hash of *entry*		

	/ Equations
1 :	e(ct1,g2) + e(g1,rinv) = 0 / encryption of *addr*
2 :	e(ct2,g2) + e(ek,rinv) - e(addr,g2) = 0
3 :	e(pk,g2) + e(g1,sk) = 0 / knowledge of sk_U
4 :	e(g1,vk11) + e(pk,vk21) + e(addr,vk31) = 0 / valid TSIG signature
5 :	e(fsig11,ppA11) + e(fsig12,ppA21) = 0
6 :	e(ssig11,ppUA11) + e(ssig12,ppUA21) = 0
7 :	e(tsig11, ppVA11) + e(tsig12,ppVA21) = 0
8 :	e(tsig11, ppVA11) + e(tsig12,ppVA21) + e(ssig11,ppUA11) + e(ssig12,ppUA21) + e(g1,vk11) + e(addr,vk21) + e(pk,vk31) - e(fsig11,ppA11) - e(fsig12,ppA21) = 0
9 :	e(ssig11, fosig) - e(tsig11,g2) = 0
10 :	e(ssig12, fosig) - e(tsig12,g2) = 0

Figure B.2.: The concrete equations to implement $\mathcal{R}_{\mathsf{CE}}$.

B.2. Security Proof

We now provide the formal proof of Theorem 4.3.1, i.e., of the UC-security of our protocol Π_{POBA}. First, we re-state the theorem formally:

Theorem B.2.1 (formal). *If* ZK *is a straight-line simulation-extractable non-interactive zero-knowledge proof system,* SIG *is an EUF-CMA-secure signature scheme,* TSIG *is a TS-UF-1-secure threshold signature scheme, and* TPKE *is an IND-CPA-secure threshold PKE, then* Π_{POBA} *UC-realizes* $\mathcal{F}$ *in the* $\{\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{Keygen}}^{\text{TPKE}}, \mathcal{F}_{\text{Keygen}}^{\text{TSIG}}, \mathcal{F}_{\text{SMT}}, \mathcal{F}_{\text{SMT}}^{\text{os-auth}}, \mathcal{F}_{MPC(\cdot)}, \mathcal{F}_{MPC(\cdot)}^{ext}\}$*-hybrid model wrt. adversaries* $\mathcal{A}$ *that may statically corrupt any number of users maliciously and up to* $t \leq \frac{n-1}{2}$ *operators passively.*

To prove Theorem B.2.1, we first provide a simulator $\mathcal{S}$ and then show indistinguishability to the real protocol through a series of games.

Simulator $\mathcal{S}$

State of the Simulator

- Setup
  - Common reference string crs_{ZK}
  - Signature parameters crs_{TSIG}
  - Trapdoor td_{ZK} for extracting and simulating zero-knowledge proofs
  - Consortium encryption key ek and decryption key shares $\{\text{vk}_i, \text{dk}_i\}_{i\in[n]}$
  - Consortium verification key vk and signing key shares $\{\text{vk}_i, sk_i\}_{i\in[n]}$
- Global state
  - A list $\text{L}_{\text{Free}}^{\mathcal{S}}$ of free logbook addresses
  - The logbook index $\text{L}_{\text{Keys}}^{\mathcal{S}}$ that maps user public keys to logbook addresses
  - The memory M_j for each period j
- For each honest operator:
  - Its signature keypair $(\text{sk}_O, \text{vk}_O)$
  - A list $\text{L}_{\text{Pending}}$ of pending entries
  - A value id_{last} for indexing pending entries
- For each semi-honest operator:
  - Its signature keypair $(\text{sk}_O, \text{vk}_O)$
  - A share of the consortium signing key $(\text{vk}_i, \text{sk}_i)$
  - A share of the consortium decryption key $(\text{vk}_i, \text{dk}_i)$
  - A list $\text{L}_{\text{Pending}}$ of pending entries
  - A value id_{last} for indexing pending entries
  - A sharing of the oblivious memory $\text{Share}M_j$ for each period j
  - A list L_{Free} of free logbook addresses
  - The logbook index L_{Keys} that maps user public keys to logbook addresses
- For each honest user:
  - Verification key of home operator vk_{HO}^{U}
  - Credentials $creds_U \coloneqq (\text{sk}_U, \text{pk}_U, addr, \sigma)$
  - List $\text{L}_{\text{Receipt}}^{\text{U}}$ of receipts for logbook entries

Simulation of $\mathcal{F}_{\text{CRS}}$:
A corrupted party can issue calls to $\mathcal{F}_{\text{CRS}}$ whenever it wants. Since simulation of $\mathcal{F}_{\text{CRS}}$ is straightforward, we specify here once how all calls from corrupted U and semi-honest O to $\mathcal{F}_{\text{CRS}}$ are handled and omit calls to $\mathcal{F}_{\text{CRS}}$ in the following description of the simulator.

- Upon receiving `(value)` from a party P to $\mathcal{F}_{\text{CRS}}$:
  1. If this is the first time such a message is received:
     a) Generate $(crs_{\text{ZK}}, td_{\text{ZK}}) \leftarrow \text{ZK.Setup}(1^\lambda)$ and store them
     b) Generate $crs_{\text{TSIG}} \leftarrow \text{TSIG.Setup}(1^\lambda)$ and store it
  2. Report message $(crs_{\text{ZK}}, crs_{\text{TSIG}})$ from $\mathcal{F}_{\text{CRS}}$ to P

Simulation of $\mathcal{F}_{\text{SMT}}$ and $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$:
The simulator simply follows the description of these functionalities.

Simulation of $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$:

- Upon receiving $(\text{Keygen}, 1^\lambda)$ from a semi-honest operator O:
  1. If $\mathsf{ek} = \bot$
     a) Run $(\mathsf{ek}, \{\mathsf{vk}_i\}_{i\in[n]}, \{\mathsf{dk}_i\}_{i\in[n]}) \leftarrow \mathsf{TPKE.Keygen}(1^\lambda, n, t)$
     b) Store ek and $\{\mathsf{vk}_i, \mathsf{dk}_i\}_{i\in[n]}$
  2. Retrieve ek and $(\mathsf{vk}_i, \mathsf{dk}_i) \in \{\mathsf{vk}_i, \mathsf{dk}_i\}_{i\in[n]}$ for O
  3. After receiving input from all operators, output $(\mathsf{ek}, \mathsf{vk}_i, \mathsf{dk}_i)$ to O

Simulation of $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$:

- Upon receiving $(\text{Keygen}, crs_{\text{TSIG}}')$ from a semi-honest operator O:
  1. If $\mathsf{vk} = \bot$
     a) Run $(\mathsf{vk}, \{\mathsf{vk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]}) \leftarrow \mathsf{TSIG.Keygen}(crs_{\text{TSIG}}, n, t)$
     b) Store vk and $\{\mathsf{vk}_i, \mathsf{sk}_i\}_{i\in[n]}$
  2. Retrieve vk and $(\mathsf{vk}_i, \mathsf{sk}_i) \in \{\mathsf{vk}_i, \mathsf{sk}_i\}_{i\in[n]}$ for O
  3. After receiving input from all operators, output $(\mathsf{vk}, \mathsf{vk}_i, \mathsf{sk}_i)$

Init (honest O):

- Upon receiving $(\texttt{init}, O)$ from $\mathcal{F}_{\text{POBA}}$:
  1. Record input from O to $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$ and $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$
  2. Wait for the simulations of $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$ and $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$ to deliver output to O
  3. If $M_0 = \bot$, initialize the memory for period 0
  4. Generate and store $(\mathsf{sk}_O, \mathsf{vk}_O) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  5. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output

Init (semi-honest O):

- The simulator executes the protocol honestly, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages to other parties. If the adversary did not make the protocol abort, allow $\mathcal{F}_{\text{POBA}}$ to deliver output.

UReg (honest U, honest HO):

- Upon receiving $((\texttt{ureg}, ssid, pid_U), HO)$ from $\mathcal{F}_{\text{POBA}}$:
  1. Report output $(ssid\|1, pid_{HO}, (\mathsf{ek}, \mathsf{vk}, \mathsf{vk}_O))$ from $\mathcal{F}_{\text{SMT}}$ to U
- When the adversary allows delivery of that output:
  1. Wait until receiving $((\texttt{ureg}, ssid, pid_{HO}), U)$ from $\mathcal{F}_{\text{POBA}}$ (if it has not yet been received)
  2. Store vk_O as vk_{HO}^U
  3. Generate and store $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  4. Generate $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt \coloneqq \mathsf{pk}_U, td_{\text{ZK}})$
  5. Report output $(ssid\|2, pid_U, (\mathsf{pk}_U, \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$ to O
- When the adversary allows delivery of that output:
  1. Get next free logbook address $addr \leftarrow \mathsf{L}_{\mathsf{Free}}.\mathsf{pop}()$
  2. Add $(\mathsf{pk}_U, \bot, addr, U)$ to $\mathsf{L}_{\text{Keys}}^{\mathcal{S}}$
  3. Report a message $(ssid, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ from HO to each other operator O'
- For each semi-honest O', when the adversary allows delivery of that message:
  1. Wait until receiving $((\texttt{ureg}, ssid), O')$ from $\mathcal{F}_{\text{POBA}}$ (if it has not been received yet)
  2. Execute the protocol honestly for O', sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- For each honest O', when the adversary allows delivery of that message:
  1. Wait until receiving $((\texttt{ureg}, ssid), O')$ from $\mathcal{F}_{\text{POBA}}$ (if it has not been received yet)
  2. Set $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\text{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$ where i is the index of O'
  3. Report a message of length $|(ssid, \sigma_i)|$ from O' to HO
- After the adversary allowed delivery of all messages from O' to HO:
  1. Set T to be the first $t+1$ partial signatures σ_i
  2. $\sigma \leftarrow \mathsf{TSIG.Combine}(crs_{\text{TSIG}}, T)$
  3. Store $(\mathsf{pk}_U, \bot)$ in $\mathsf{L}_{\text{Accounts}}^{\text{HO}}$ and $creds_U \coloneqq (\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  4. Report output $(ssid\|3, pid_{HO}, (addr, \sigma))$ from $\mathcal{F}_{\text{SMT}}$ to U
  5. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output to HO
- After the adversary allowed delivery of the output of $\mathcal{F}_{\text{SMT}}$ to U:
  1. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output to U

UReg (honest U, semi-honest HO):

- The simulator executes the protocol honestly for HO, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages to U.
- When the adversary allows delivery of the output $(ssid\|1,\ pid_{HO},\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ from $\mathcal{F}_{\text{SMT}}$ to U:
  1. Wait until receiving $((\texttt{ureg},\ ssid,\ pid_{HO}),\ U)$ from $\mathcal{F}_{\text{POBA}}$ (if it has not been received yet)
  2. Generate and store $(\mathsf{sk}_U,\ \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  3. Store vk_O as vk_{HO}^U
  4. $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}},\ \mathcal{R}_{\text{UR}},\ stmt \coloneqq (\mathsf{pk}_U),\ td_{\text{ZK}})$
  5. Report output $(ssid\|2,\ pid_U,\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$ to HO
- For each O', when the adversary allows delivery of the message $(ssid,\ \mathsf{pk}_U,\ \pi_{\mathsf{sk}_U},\ addr,\ U)$ from HO:
  1. Wait until receiving $((\texttt{ureg},\ ssid),\ O')$ from $\mathcal{F}_{\text{POBA}}$ (if it has not been received yet)
  2. For the first O':
     a) Remove $addr$ from L_{Free}
     b) Add $(\mathsf{pk}_U,\ \bot,\ addr,\ U)$ to $\mathsf{L}^{\mathcal{S}}_{\text{Keys}}$
  3. Execute the protocol honestly for O'
  4. If O' is semi-honest, send the party's state to the adversary after each activation and wait for `continue` from the adversary before sending out any messages.
- When the adversary allows delivery of the output $(ssid\|3,\ pid_{HO},\ (addr,\ \sigma))$ from $\mathcal{F}_{\text{SMT}}$ to U:
  1. If $\mathsf{SIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$, abort
  2. Store $creds_U \coloneqq (\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  3. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output to U

UReg (corrupted U):

- The simulator executes the protocol honestly, sending each semi-honest party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages from it.
- Additionally, when receiving $(ssid,\ \mathsf{pk}_U,\ \pi_{\mathsf{sk}_U})$ from U, if the proof verifies successfully:
  1. Extract $wit \coloneqq (\mathsf{sk}_U) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}},\ \mathcal{R}_{\text{UR}},\ stmt,\ \pi_{\mathsf{sk}_U},\ td_{\text{ZK}})$
  2. Store $(\mathsf{pk}_U,\ \mathsf{sk}_U,\ addr,\ U)$ in $\mathsf{L}^{\mathcal{S}}_{\text{Keys}}$
  3. Send $(\texttt{ureg},\ ssid,\ pid_{HO})$ to $\mathcal{F}_{\text{POBA}}$ on behalf of U
- When simulation of HO sends the message $(ssid\|3,\ pid_U,\ (addr,\ \sigma))$ to $\mathcal{F}_{\text{SMT}}$, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to HO

CreateEntry (honest U, honest O):

- Simply report messages of the appropriate length.

CreateEntry (honest U, semi-honest O):

- The simulator runs the protocol honestly for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages from it.
- Upon receiving $((\texttt{createentry},\ ssid),\ \texttt{user})$ and $((\texttt{createentry},\ ssid,\ period,\ entry),\ O)$ from $\mathcal{F}_{\text{POBA}}$:
  1. Set $addr \coloneqq 0$
  2. Generate $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek},\ addr; r)$
  3. Set $h \coloneqq H(period,\ ct_U,\ entry)$
  4. Set $stmt \coloneqq (ct_U,\ \mathsf{ek},\ crs_{\text{TSIG}},\ \mathsf{vk},\ h)$
  5. Generate $\pi \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}},\ \mathcal{R}_{\text{CE}},\ stmt,\ td_{\text{ZK}})$
  6. Report output $(\texttt{send},\ ssid\|1,\ (ct_U,\ \pi))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ to O
- When the adversary allows delivery of output $(\texttt{respond},\ ssid\|1,\ (\sigma_{entry},\ \mathsf{vk}_O))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$:
  1. Store $(entry,\ ct_U,\ r,\ \sigma_{entry},\ \mathsf{vk}_O)$ in $\mathsf{L}^{\text{U}}_{\text{Receipt}}$
  2. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output

CreateEntry (corrupted U, semi-honest O):

- The simulator executes the protocol honestly for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- Additionally, the simulator does the following:
  - When the adversary allows delivery of output $(\texttt{send},\ ssid\|1,\ (entry_U,\ ct_U,\ \pi))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$, if the honest protocol did not abort as a result of this message:
    1. Get $wit \coloneqq (\mathsf{pk}_U,\ \mathsf{sk}_U,\ addr,\ r,\ \sigma) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}},\ \mathcal{R}_{\text{CE}},\ stmt,\ \pi,\ td_{\text{ZK}})$
    2. Find $(\mathsf{pk}_U,\ \mathsf{sk}_U,\ addr,\ U^*)$ in $\mathsf{L}^{\mathcal{S}}_{\text{Keys}}$, abort if no such entry exists
    3. If U^* is honest, abort
    4. Send $(\texttt{createentry},\ ssid,\ period,\ entry_U)$ to $\mathcal{F}_{\text{POBA}}$ on behalf of user U^*

CreateEntry (corrupted U, honest O):
- After receiving $((\texttt{createentry}, ssid), O)$ from $\mathcal{F}_{\text{POBA}}$ and the adversary allowing delivery of the output $(\texttt{send}, ssid\|1, (entry_U, ct_U, \pi))$ from $\mathcal{F}_{\text{SMT}}$ to O:
  1. $h \coloneqq H(period, ct_U, entry_U)$, $stmt \coloneqq (ct_U, \mathsf{ek}, crs_{\text{TSIG}}, \mathsf{vk}, h)$
  2. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\mathsf{CE}}, stmt, \pi) \neq 1$, ignore this message
  3. Otherwise, get $wit \coloneqq (\mathsf{pk}_U, \mathsf{sk}_U, addr, r, \sigma) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}}, \mathcal{R}_{\mathsf{CE}}, stmt, \pi, td_{\text{ZK}})$, abort if this fails
  4. Find $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$, abort if no such entry exists
  5. If U^* is honest, abort
  6. Send $(\texttt{createentry}, ssid, period, entry_U, pid_O)$ to $\mathcal{F}_{\text{POBA}}$ on behalf of user U^*

InsertEntry:
- Upon receiving $((\mathsf{insert}, ssid, period, id), O)$ from $\mathcal{F}_{\text{POBA}}$ (i.e., O is semi-honest):
  1. Honestly execute the protocol for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- Upon receiving $((\mathsf{insert}, ssid), O)$ and $((\mathsf{insert}, ssid, period, id), O)$ from $\mathcal{F}_{\text{POBA}}$ (i.e., O is honest):
  1. Find $(id, period, ct_U, entry, \pi) \in \mathsf{L}^{\mathsf{O}}_{\mathsf{Pending}}$ and ignore this message if no entry with $(id, period)$ exists
  2. Send $(ssid, ct_U, entry, \pi)$ to all O'
  3. Honestly run the checks from the real protocol
  4. Treat O as having provided input to $\mathcal{F}_{\text{MPC}(\mathsf{insert})}$
- Upon receiving $((\mathsf{insert}, ssid), O')$ from $\mathcal{F}_{\text{POBA}}$ for honest O':
  1. Wait until the adversary allowed delivery of the message $(ct_U, entry, \pi)$ from O
  2. Honestly run the checks from the real protocol
  3. Treat O' as having provided input to $\mathcal{F}_{\text{MPC}(\mathsf{insert})}$
- Upon receiving $((\mathsf{insert}, ssid), O')$ from $\mathcal{F}_{\text{POBA}}$ for semi-honest O':
  1. Honestly execute the protocol for O', sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- Simulate $\mathcal{F}_{\text{MPC}(\mathsf{insert})}$ as follows:
  - Upon receiving $(\mathsf{Run}, \mathsf{Share}M, \mathsf{Share}x)$ from all semi-honest operators and having treated all honest operators as having provided input to $\mathcal{F}_{\text{MPC}(\mathsf{insert})}$:
    1. Chose a random $addr^* \in \mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
    2. Ignore $\mathsf{Share}M$ and $\mathsf{Share}x$, use M and partial decryptions of $\mathsf{TPKE.Encrypt}(\mathsf{ek}, addr^*)$ instead
    3. Follow the description of insert honestly
    4. When delivering output to semi-honest operators, instead of $\mathsf{Share}M$ use $\mathsf{Share}0$, where 0 is a memory the same size as M but consisting only of 0-entries
- When the adversary allows delivery of output from $\mathcal{F}_{\text{MPC}(\mathsf{insert})}$ to an operator O, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O

ProveEntry (honest U, honest O):
- Simply report a message of the appropriate length from U to O and allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O after the adversary allowed that message to be delivered

ProveEntry (honest U, semi-honest O):
- The simulator executes the protocol for O honestly, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- When $\mathcal{F}_{\text{POBA}}$ wants to deliver output $(ssid, \texttt{proveentry}, pid_U, entry, period)$ to O:
  1. Retrieve $(entry, ct_U, r, \sigma_{entry}, \mathsf{vk}_O)$ from $\mathsf{L}^{\mathsf{U}}_{\mathsf{Receipt}}$ and $(\mathsf{pk}_U, \perp, addr, U)$ from $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  2. Set $stmt \coloneqq (\mathsf{pk}_U, addr, ct_U)$ and $\pi \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}}, \mathcal{R}_{\mathsf{PE}}, stmt, td_{\text{ZK}})$
  3. Report output $(ssid\|1, pid_U, (period, entry, \mathsf{pk}_U, addr, \sigma, ct_U, \sigma_{entry}, \mathsf{vk}_O, \pi))$ to O from $\mathcal{F}_{\text{SMT}}$
- When the adversary allows O to generate output, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O

ProveEntry (corrupted U)
- When the adversary allows delivery of output $(ssid\|1, pid_U, (period, entry, \mathsf{pk}_U, addr, \sigma, ct_U, \sigma_{entry}, \mathsf{vk}_O, \pi))$ from $\mathcal{F}_{\text{SMT}}$:
  1. Follow the steps of the honest protocol, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages if O is semi-honest
  2. Before asking the adversary for permission to deliver output:
     a) Set $stmt \coloneqq (\mathsf{pk}_U, addr, ct_U)$
     b) Get $wit \coloneqq (r, \mathsf{sk}_U) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}}, \mathcal{R}_{\mathsf{PE}}, stmt, \pi, td_{\text{ZK}})$, abort if this fails
     c) Find the entry $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$, abort if no matching entry exists
     d) If U^* is honest, abort
     e) Send message $(\texttt{proveentry}, ssid, entry, period, O)$ to $\mathcal{F}_{\text{POBA}}$ on behalf of U^*
  3. When the adversary allows O to deliver output, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O

Analytics:
- If all operators are honest, simply report messages for revealing output shares of the appropriate lengths.
- Otherwise (at least one operator is semi-honest), upon receiving $((\texttt{analytics}, \mathit{ssid}, \mathit{func}, \{\mathit{period}_i\}, \{U_i\}, \mathit{aux}_{O'}), O')$ from $\mathcal{F}_{\text{POBA}}$ (i.e., O' is semi-honest):
  1. Simulate O' honestly, sending the state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
  2. When the adversary allows generation of output, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O'
- Upon receiving $((\texttt{analytics}, \mathit{ssid}), O)$ from $\mathcal{F}_{\text{POBA}}$ (i.e., O is honest):
  1. Do nothing
- Wait until receiving $(\texttt{analytics}, \mathit{ssid}, \mathit{sequence})$ and outputs $(\mathit{ssid}, \texttt{result}_{O'})$ for each semi-honest operator O', then simulate $\mathcal{F}^{\text{ext}}_{\text{MPC}(\mathit{func})}$ as follows:
  1. For each honest O^*, set $\texttt{result}_{O^*} \coloneqq 0$
  2. Set $\texttt{result} \coloneqq \bigcup \texttt{result}_O$ (this is 0 for honest operators, and the output of $\mathcal{F}_{\text{POBA}}$ for semi-honest operators)
  3. Output shares $\text{Share}(\text{stop}, \texttt{result}), \{\text{Share}M_i\}, \mathit{round}, \mathit{sequence}$ to all parties
- When the simulation of $\mathcal{F}^{\text{ext}}_{\text{MPC}(\mathit{func})}$ has delivered output to honest O:
  1. For each semi-honest O', reveal $\text{Share}\texttt{result}_{O'}$ to O'
  2. For each honest O^*, report a message of the correct length from O to O^*
- When the adversary allowed delivery of all share reveals towards an honest operator O:
  1. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O

We proceed in a series of games, starting with the real experiment, which we gradually transform into the ideal experiment. In these games, $\mathcal{F}^{i+1}$ behaves the same as $\mathcal{F}^i$ except for the stated changes, and $\mathcal{S}^{i+1}$ behaves the same as $\mathcal{S}^i$ except for the stated changes.

Game 0. Hybrid 0 is the real experiment. That is,

$$H_0 \coloneqq \text{Exp}^{\Pi_{\text{POBA}}}_{\mathcal{F}_{\text{CRS}}, \mathcal{F}^{\text{TPKE}}_{\text{Keygen}}, \mathcal{F}^{\text{TSIG}}_{\text{Keygen}}, \mathcal{F}_{\text{MPC}(\cdot)}, \mathcal{F}_{\text{MPC}}, \mathcal{S}^0, \text{Enc}}$$

with $\mathcal{S}^0$ being the dummy adversary and all parties executing the real protocol.

Game 1. In hybrid 1, the simulator $\mathcal{S}^1$ assumes control of $\mathcal{F}_{\text{CRS}}$, $\mathcal{F}^{\text{TPKE}}_{\text{Keygen}}$, $\mathcal{F}^{\text{TSIG}}_{\text{Keygen}}$, and $\mathcal{F}_{\text{MPC}(\cdot)}$ but still executes them honestly. We also introduce $\mathcal{F}^1$ (see Fig. B.3) and all honest parties are replaced by dummy parties that forward their inputs to $\mathcal{F}^1$ and when receiving output from $\mathcal{F}^1$ forward it to the environment. $\mathcal{F}^1$, upon receiving any message from a dummy party, forwards it to the simulator $\mathcal{S}^1$ and asks it for output. $\mathcal{S}^1$ simply executes the real protocol for all honest parties using the inputs received from $\mathcal{F}^1$ and instructs $\mathcal{F}^1$ to deliver the resulting outputs.

$\mathcal{F}^1$

- Upon receiving some input (msg) from party P:
  1. Send $(\texttt{in}, P, \text{msg})$ to the simulator
- Upon receiving $(\texttt{out}, P, \text{msg})$ from the simulator:
  1. Send (msg) to party P

Figure B.3.: $\mathcal{F}^1$

Proof Sketch: Game 0 ≈ Game 1.
$\mathcal{S}^1$ executes the same code as the real parties on the same inputs. Thus, Game 0 and Game 1 are identical from the environment's view.

Game 2. In this game, $\mathcal{S}^2$ generates the common reference string for the zero-knowledge proof system with an extraction and simulation trapdoor td_{ZK} as $(crs_{\text{ZK}}, td_{\text{ZK}}) \leftarrow \text{ZK.Setup}(1^\lambda)$.

Handling of $(\texttt{init})$ in $\mathcal{F}^2$

- Upon receiving input $(\texttt{init})$ from operator O:
  1. Send $(\texttt{in}, O, (\texttt{init}))$ to the simulator.
- Upon receiving $(\texttt{out}, O, (\texttt{init}, \texttt{ok}))$ from the simulator,
  1. Send $(\texttt{init}, \texttt{ok})$ to O.

Handling of $(\texttt{init})$ in $\mathcal{F}^3$

- Upon receiving input $(\texttt{init})$ from operator O:
  1. Send $((\texttt{init}), O)$ to the simulator
  2. After receiving input from all $O \in \mathcal{O}$, set *initialized* $\coloneqq 1$
  3. Deliver private delayed output $(\texttt{init}, \texttt{ok})$ to O
- Upon receiving $(\texttt{out}, O, (\texttt{init}, \texttt{ok}))$ from the simulator:
  1. Ignore this message.

Figure B.4.: Changes to $\mathcal{F}_{\text{POBA}}$

Handling of $(\texttt{init})$ in $\mathcal{S}^2$

Upon receiving $(\texttt{in}, O, (\texttt{init}))$ from $\mathcal{F}^2$:
1. Run the protocol honestly for O
2. When the protocol would generate output $(\texttt{init}, \texttt{ok})$ for O, send $(\texttt{out}, O, (\texttt{init}, \texttt{ok}))$ to $\mathcal{F}^2$

Handling of $(\texttt{init})$ in $\mathcal{S}^3$

Upon receiving $((\texttt{init}), O)$ from $\mathcal{F}^3$:
- If O is semi-honest:
  1. Run the protocol honestly, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages to other parties. If the adversary did not make the protocol abort, allow $\mathcal{F}^3$ to deliver output.
- If O is honest:
  1. Record input from O to $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$ and $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$
  2. Wait for the simulations of $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$ and $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$ to deliver output to O
  3. If $M_0 = \bot$, initialize the memory for period 0
  4. Store $(\mathsf{sk}_O, \mathsf{vk}_O) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  5. Allow $\mathcal{F}^3$ to deliver output

Figure B.5.: Changes to $\mathcal{S}$

Proof Sketch: Game 1 ≈ Game 2.
Indistinguishability follows from the simulation-extractability of ZK.

Game 3. In this game, $\mathcal{F}^3$ handles $(\texttt{init})$ inputs the same way as $\mathcal{F}_{\text{POBA}}$ and $\mathcal{S}^3$ handles them the same way as $\mathcal{S}$. The changes to $\mathcal{F}^3$ are detailed in Fig. B.4 and to $\mathcal{S}^3$ in Fig. B.5.

Proof Sketch: Game 2 ≈ Game 3.
Simulation for semi-honest operators is identical between these games (only the format of the message from $\mathcal{F}^3$ changed). The only visible effect of honest operators is $\mathcal{F}_{\text{Keygen}}^{\text{TPKE}}$ and $\mathcal{F}_{\text{Keygen}}^{\text{TSIG}}$ delivering output to semi-honest operators only after all honest operators have called it.

Game 4. In this game, $\mathcal{F}^4$ handles messages $(\texttt{ureg}, \ldots)$ the way $\mathcal{F}_{\text{POBA}}$ does if both the user and the home operator involved are honest. The changes to $\mathcal{S}^4$ are detailed in Fig. B.6.

Proof Sketch: Game 3 ≈ Game 4.
The only change visible to the environment between these two games is how π_{sk_U} is generated, the lengths of messages between honest parties and the other parts of the messages to semi-honest parties are identical in both games. Thus, the probability of

UReg in $\mathcal{S}^3$, honest U, honest HO

- Upon receiving $(\texttt{in}, HO, (\texttt{ureg}, pid_U, ssid))$ from $\mathcal{F}^3$, perform as HO:
  1. Send message $(ssid\|1,\ pid_U,\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ to $\mathcal{F}_{\text{SMT}}$
- Upon receiving $(\texttt{in},\ U,\ (\texttt{ureg},\ ssid,\ pid_{HO}))$ from $\mathcal{F}^3$, perform as U:
  1. Wait until $(ssid\|1,\ pid_{HO},\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ was received from $\mathcal{F}_{\text{SMT}}$
  2. Store ek, vk and vk_O as vk_{HO}
  3. Obtain and store crs_{ZK} and crs_{TSIG} from $\mathcal{F}_{\text{CRS}}$
  4. $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  5. $stmt \coloneqq (\mathsf{pk}_U)$, $wit \coloneqq (\mathsf{sk}_U)$
  6. $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, wit)$
  7. Send message $(ssid\|2,\ pid_{HO},\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ to $\mathcal{F}_{\text{SMT}}$
- After delivering output $(ssid\|2,\ pid_U,\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$ to HO, perform as HO:
  1. $stmt \coloneqq (\mathsf{pk}_U)$
  2. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ abort
  3. Get next free logbook address $addr \leftarrow \mathsf{L}_{\mathsf{Free}}.\mathsf{pop}()$
  4. Add $(\mathsf{pk}_U, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}$
  5. Send message $(ssid, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ to each O'
- Upon receiving $(\texttt{in}, O', (\texttt{ureg}, ssid))$ from $\mathcal{F}^3$, perform as O' (for each O'):
  1. Wait until $(ssid, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ was sent from HO to O'
  2. $stmt \coloneqq (\mathsf{pk}_U)$
  3. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ or $addr \notin \mathsf{L}_{\mathsf{Free}}$ abort
  4. Remove $addr$ from $\mathsf{L}_{\mathsf{Free}}$, add $(\mathsf{pk}_U, addr, U)$ to $\mathsf{L}_{\mathsf{Keys}}$
  5. $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\text{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  6. Send message $(ssid, \sigma_i)$ to HO
- After all messages $(ssid, \sigma_i)$ from honest and semi-honest O' have been received by HO, perform as HO:
  1. $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\text{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  2. Set T to be the first valid $t+1$ partial signatures σ_i
  3. $\sigma \leftarrow \mathsf{TSIG.Combine}(crs_{\text{TSIG}}, T)$
  4. Store (pk_U) in $\mathsf{L}^{\text{HO}}_{\text{Accounts}}$
  5. Send message $(ssid\|3,\ pid_U,\ (addr,\ \sigma))$ to $\mathcal{F}_{\text{SMT}}$
  6. Send $(\texttt{out}, HO, (\texttt{ok}))$ to $\mathcal{F}^3$
- After delivering output $(ssid\|3,\ pid_{HO},\ (addr,\ \sigma))$ from $\mathcal{F}_{\text{SMT}}$ to U, perform as U:
  1. If $\mathsf{TSIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$, abort
  2. Store $(\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  3. Send $(\texttt{out}, U, (\texttt{ok}))$ to $\mathcal{F}^3$

UReg in $\mathcal{S}^4$, honest U, honest HO

- Upon receiving $((\texttt{ureg}, ssid, pid_U), HO)$ from $\mathcal{F}^4$:
  1. Report that $\mathcal{F}_{\text{SMT}}$ wants to generate output $(ssid\|1,\ pid_{HO},\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ to U
- When the adversary allows delivery of that output:
  1. Wait until receiving $((\texttt{ureg}, ssid, pid_{HO}), U)$ from $\mathcal{F}^4$ (if it has not yet been received)
  2. Store vk_O as vk^U_{HO}
  3. Generate and store $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \mathsf{SIG.Keygen}(1^\lambda)$
  4. Generate $\pi_{\mathsf{sk}_U} \leftarrow \mathsf{ZK.Sim}(crs_{\text{ZK}},\ \mathcal{R}_{\text{UR}},\ stmt \coloneqq (\mathsf{pk}_U),\ td_{\text{ZK}})$
  5. Report that $\mathcal{F}_{\text{SMT}}$ wants to generate output $(ssid\|2,\ pid_U,\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ to O
- When the adversary allows delivery of that output:
  1. Get next free logbook address $addr \leftarrow \mathsf{L}_{\mathsf{Free}}.\mathsf{pop}()$
  2. Add $(\mathsf{pk}_U, \bot, addr, U)$ to $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  3. Report a message $(ssid,\ \mathsf{pk}_U,\ \pi_{\mathsf{sk}_U},\ addr,\ U)$ from HO to each other operator O'
- For each semi-honest O', when the adversary allows delivery of that message:
  1. Wait until receiving $((\texttt{ureg}, ssid), O')$ from $\mathcal{F}^4$ (if it has not been received yet)
  2. Execute the protocol honestly for O', sending the party's state to the adversary after each activation and waiting for $\texttt{continue}$ from the adversary before sending out any messages.
- For each honest O', when the adversary allows delivery of that message:
  1. Wait until receiving $((\texttt{ureg}, ssid), O')$ from $\mathcal{F}^4$ (if it has not been received yet)
  2. Set $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\text{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$, where i is the index of O'
  3. Report message $(ssid,\ \sigma_i)$ from O' to HO
- After the adversary allowed delivery of all messages from O' to HO:
  1. Set T to be the first $t+1$ σ_i
  2. $\sigma \leftarrow \mathsf{TSIG.Combine}(crs_{\text{TSIG}}, T)$
  3. Store $(\mathsf{pk}_U, \bot)$ in $\mathsf{L}^{\text{HO}}_{\text{Accounts}}$ and $creds_U \coloneqq (\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  4. Report that $\mathcal{F}_{\text{SMT}}$ wants to generate output $(ssid\|3,\ pid_{HO},\ (addr,\ \sigma))$ to U
  5. Allow $\mathcal{F}^4$ to deliver output to HO
- After the adversary allowed delivery of the output from $\mathcal{F}_{\text{SMT}}$ to U:
  1. Allow $\mathcal{F}^4$ to deliver output to U

Figure B.6.: Handling registration for honest user and home operator

the environment to distinguish these games is bounded by the probability to break zero-knowledge of ZK, and hence these games are indistinguishable.

Game 5. In this game, $\mathcal{F}^5$ handles messages $(\texttt{ureg}, \ldots)$ the way $\mathcal{F}_{\text{POBA}}$ does if the user involved is honest but the home operator is semi-honest. The changes to $\mathcal{S}^5$ are detailed in Fig. B.7.

Proof Sketch: Game 4 ≈ Game 5.
Again, the only change visible to the environment is the generation of π_{sk_U}, all other messages are the same in both games. Thus, the probability of the environment to distinguish

UReg in $\mathcal{S}^4$, honest U,semi-honest HO

- Upon receiving $(\texttt{in}, HO, (\texttt{ureg}, pid_U, ssid))$ from $\mathcal{F}^4$, perform as HO:
  1. Send message $(ssid\|1, pid_U, (\mathsf{ek}, \mathsf{vk}, \mathsf{vk}_O))$ to $\mathcal{F}_{\text{SMT}}$
- Upon receiving $(\texttt{in}, U, (\texttt{ureg}, ssid, pid_{HO}))$ from $\mathcal{F}^4$, perform as U:
  1. Wait until $(ssid\|1, pid_{HO}, (\mathsf{ek}, \mathsf{vk}, \mathsf{vk}_O))$ was received from $\mathcal{F}_{\text{SMT}}$
  2. Store ek, vk and vk_O as vk_{HO}
  3. Obtain and store crs_{ZK} and crs_{TSIG} from $\mathcal{F}_{\text{CRS}}$
  4. $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \text{SIG.Keygen}(1^\lambda)$
  5. $stmt \coloneqq (\mathsf{pk}_U)$, $wit \coloneqq (\mathsf{sk}_U)$
  6. $\pi_{\mathsf{sk}_U} \leftarrow \text{Prove}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, wit)$
  7. Send message $(ssid\|2, pid_{HO}, (\mathsf{pk}_U, \pi_{\mathsf{sk}_U}))$ to $\mathcal{F}_{\text{SMT}}$
- After delivering output $(ssid\|2, pid_U, (\mathsf{pk}_U, \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$ to HO, perform as HO:
  1. $stmt \coloneqq (\mathsf{pk}_U)$
  2. If $\text{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ abort
  3. Get next free logbook address $addr \leftarrow \mathsf{L}_{\text{Free}}.\text{pop}()$
  4. Add $(\mathsf{pk}_U, addr, U)$ to L_{Keys}
  5. Send message $(ssid, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ to each O'
- Upon receiving $(\texttt{in}, O', (\texttt{ureg}, ssid))$ from $\mathcal{F}^4$, perform as O' (for each O'):
  1. Wait until $(ssid, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ was sent from HO to O'
  2. $stmt \coloneqq (\mathsf{pk}_U)$
  3. If $\text{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ or $addr \notin \mathsf{L}_{\text{Free}}$ abort
  4. Remove $addr$ from L_{Free}, add $(\mathsf{pk}_U, addr, U)$ to L_{Keys}
  5. $\sigma_i \leftarrow \text{TSIG.PartSign}(crs_{\text{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  6. Send message $(ssid, \sigma_i)$ to HO
- After all messages $(ssid, \sigma_i)$ from honest and semi-honest O' have been received by HO, perform as HO:
  1. $\sigma_i \leftarrow \text{TSIG.PartSign}(crs_{\text{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  2. Set T to be the first valid $t+1$ partial signatures σ_i
  3. $\sigma \leftarrow \text{TSIG.Combine}(crs_{\text{TSIG}}, T)$
  4. Store (pk_U) in $\mathsf{L}^{\text{HO}}_{\text{Accounts}}$
  5. Send message $(ssid\|3, pid_U, (addr, \sigma))$ to $\mathcal{F}_{\text{SMT}}$
  6. Send $(\texttt{out}, HO, (\texttt{ok}))$ to $\mathcal{F}^4$
- After delivering output $(ssid\|3, pid_{HO}, (addr, \sigma))$ from $\mathcal{F}_{\text{SMT}}$ to U, perform as U:
  1. If $\text{TSIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$, abort
  2. Store $(\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  3. Send $(\texttt{out}, U, (\texttt{ok}))$ to $\mathcal{F}^4$

UReg in $\mathcal{S}^5$, honest U, semi-honest HO

- The simulator executes the protocol honestly for HO, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages to U.
- When the adversary allows delivery of output $(ssid\|1, pid_{HO}, (\mathsf{ek}, \mathsf{vk}, \mathsf{vk}_O))$ to U:
  1. Wait until receiving $((\texttt{ureg}, ssid, pid_{HO}), U)$ from $\mathcal{F}^5$ (if it has not yet been received)
  2. Store vk_O as vk^U_{HO}
  3. Generate and store $(\mathsf{sk}_U, \mathsf{pk}_U) \leftarrow \text{SIG.Keygen}(1^\lambda)$
  4. Generate $\pi_{\mathsf{sk}_U} \leftarrow \text{ZK.Sim}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt \coloneqq (\mathsf{pk}_U), td_{\text{ZK}})$
  5. Report that $\mathcal{F}_{\text{SMT}}$ wants to generate output $(ssid\|2, pid_U, (\mathsf{pk}_U, \pi_{\mathsf{sk}_U}))$ to O
- For each O', when the adversary allows delivery of the message $(ssid, \mathsf{pk}_U, \pi_{\mathsf{sk}_U}, addr, U)$ from HO:
  1. Wait until receiving $((\texttt{ureg}, ssid), O')$ from $\mathcal{F}^5$ (if it has not been received yet)
  2. For the first O':
     a) Remove $addr$ from L_{Free}
     b) Add $(\mathsf{pk}_U, \bot, addr, U)$ to $\mathsf{L}^{\mathcal{S}}_{\text{Keys}}$
  3. Execute the protocol honestly for O'
  4. If O' is semi-honest, send the party's state to the adversary after each activation and wait for `continue` from the adversary before sending out any messages.
- After the adversary allowed delivery of the output from $\mathcal{F}_{\text{SMT}}$ to U:
  1. If $\text{SIG.Vf}(\mathsf{vk}, (\mathsf{pk}_U, addr), \sigma) \neq 1$ abort
  2. Store $creds_U \coloneqq (\mathsf{sk}_U, \mathsf{pk}_U, addr, \sigma)$
  3. Allow $\mathcal{F}^5$ to deliver output to U

Figure B.7.: Handling registration for honest user and semi-honest home operator

these games is again bounded by the probability to break zero-knowledge of ZK, and hence these games are indistinguishable.

Game 6. In this game, $\mathcal{F}^6$ handles messages $(\texttt{ureg}, \ldots)$ the way $\mathcal{F}_{\text{POBA}}$ does if the user involved is corrupted. The changes to $\mathcal{S}^6$ are detailed in Fig. B.8.

Proof Sketch: Game 5 ≈ Game 6.
All messages to the user and semi-honest operators are generated honestly, thus these games only differ if $\mathcal{S}^6$ aborts because extraction of the witness failed. The probability of this is bound by the probability of breaking the simulation-extractability of ZK and hence these games are indistinguishable.

UReg in $\mathcal{S}^5$, corrupted U

- Upon receiving $(\texttt{in}, HO, (\texttt{ureg}, pid_U, ssid))$ from $\mathcal{F}^5$, perform as HO:
  1. Send message $(ssid\|1,\ pid_U,\ (\mathsf{ek},\ \mathsf{vk},\ \mathsf{vk}_O))$ to $\mathcal{F}_{\text{SMT}}$
- When the adversary allows delivery of output $(ssid\|2,\ pid_U,\ (\mathsf{pk}_U,\ \pi_{\mathsf{sk}_U}))$ from $\mathcal{F}_{\text{SMT}}$ to HO, perform as HO:
  1. $stmt \coloneqq (\mathsf{pk}_U)$
  2. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ abort
  3. Get next free logbook address $addr \leftarrow \mathsf{L}_{\text{Free}}.\text{pop}()$
  4. Add $(\mathsf{pk}_U, addr, U)$ to L_{Keys}
  5. Send message $(ssid,\ \mathsf{pk}_U,\ \pi_{\mathsf{sk}_U},\ addr,\ U)$ to each O'
- Upon receiving $(\texttt{in}, O', (\texttt{ureg}, ssid))$ from $\mathcal{F}^5$, perform as O' (for each O'):
  1. Wait until $(ssid,\ \mathsf{pk}_U,\ \pi_{\mathsf{sk}_U},\ addr,\ U)$ was delivered from HO to O'
  2. $stmt \coloneqq (\mathsf{pk}_U)$
  3. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{UR}}, stmt, \pi_{\mathsf{sk}_U}) \neq 1$ or $addr \notin \mathsf{L}_{\text{Free}}$ abort
  4. Remove $addr$ from L_{Free}, add $(\mathsf{pk}_U, addr, U)$ to L_{Keys}
  5. $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\text{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  6. Send message $(ssid, \sigma_i)$ to HO
- After all messages $(ssid, \sigma_i)$ from honest and semi-honest O' have been received by HO, perform as HO:
  1. $\sigma_i \leftarrow \mathsf{TSIG.PartSign}(crs_{\text{TSIG}}, \mathsf{sk}_i, (\mathsf{pk}_U, addr))$
  2. Set T to be the first valid $t+1$ partial signatures σ_i
  3. $\sigma \leftarrow \mathsf{TSIG.Combine}(crs_{\text{TSIG}}, T)$
  4. Store (pk_U) in $\mathsf{L}^{\text{HO}}_{\text{Accounts}}$
  5. Send message $(ssid\|3,\ pid_U,\ (addr,\ \sigma))$ to $\mathcal{F}_{\text{SMT}}$
  6. Send $(\texttt{out}, HO, (\texttt{ok}))$ to $\mathcal{F}^5$

UReg in $\mathcal{S}^6$, corrupted U

- The simulator executes the protocol honestly, sending each semi-honest party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages from it.
- Additionally, when receiving $(ssid, \mathsf{pk}_U, \pi_{\mathsf{sk}_U,})$ from U, if the proof verifies successfully:
  1. Extract $wit \coloneqq (\mathsf{sk}_U) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}},\ \mathcal{R}_{\text{UR}},\ stmt,\ \pi_{\mathsf{sk}_U},\ td_{\text{ZK}})$
  2. Store $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U)$ in $\mathsf{L}^{\mathcal{S}}_{\text{Keys}}$
  3. Send $(\texttt{ureg}, ssid, HO)$ to $\mathcal{F}^6$ on behalf of U
- When simulation of HO sends the message $(ssid\|3,\ pid_U,\ (addr,\ \sigma))$ to $\mathcal{F}_{\text{SMT}}$, allow $\mathcal{F}^6$ to deliver output to HO

Figure B.8.: Handling registration for corrupted user

$\mathcal{F}_{\text{MPC(insert)}}$ in Game 6

Parameters: A next-step function NS

- Upon receiving (Run, Share$M$, Share$x$) from all parties:
  1. Reconstruct memory M and input x
  2. Initialize $\mathsf{st}_{\text{NS}} \coloneqq (\text{start}, x)$, $data \coloneqq 0$ and $round \coloneqq 0$
  3. While $\mathsf{st}_{\text{NS}} \neq (\text{stop}, z)$, do:
     a) Run $(\mathsf{st}_{\text{NS}}, I) \leftarrow \mathsf{NS}(\mathsf{st}_{\text{NS}}, data)$
     b) If $I = (\text{read}, addr, \perp)$, set $data \coloneqq M[addr].val$
     c) Else if $I = (\text{write}, addr, val)$, set $data \coloneqq M[addr].val$ and update $M[addr].val = I.val$
     d) Update $round \coloneqq round + 1$
  4. Output shares $\text{Share}^{\text{st}}_{\text{NS}}$, Share$M$ and $round$ to all parties

$\mathcal{F}_{\text{MPC(insert)}}$ in Game 7

Parameters: A next-step function NS

- Upon receiving (Run, Share$M$, Share$x$) from all parties:
  1. Ignore Share$M$, use $M \coloneqq M_{period}$ from the simulator state instead
  2. Reconstruct input x
  3. Initialize $\mathsf{st}_{\text{NS}} \coloneqq (\text{start}, x)$, $data \coloneqq 0$ and $round \coloneqq 0$
  4. While $\mathsf{st}_{\text{NS}} \neq (\text{stop}, z)$, do:
     a) Run $(\mathsf{st}_{\text{NS}}, I) \leftarrow \mathsf{NS}(\mathsf{st}_{\text{NS}}, data)$
     b) If $I = (\text{read}, addr, \perp)$, set $data \coloneqq M[addr].val$
     c) Else if $I = (\text{write}, addr, val)$, set $data \coloneqq M[addr].val$ and update $M[addr].val = I.val$
     d) Update $round \coloneqq round + 1$
  5. Output shares $\text{Share}^{\text{st}}_{\text{NS}}$, Share$M$ and $round$ to all parties

Figure B.9.: Simulation of $\mathcal{F}_{\text{MPC(insert)}}$, Part 1

Game 7. In this game, when simulating $\mathcal{F}_{\text{MPC(insert)}}$, $\mathcal{S}^7$ ignores the input ShareM_{period} and instead uses M_{period} from its global state, see Fig. B.9.

Proof Sketch: Game 6 ≈ Game 7.
These games are identical from the environment's view since all M_i start empty and semi-honest operators always input the correct share, which corresponds to either the empty initial memory or the memory last output by $\mathcal{F}_{\text{MPC(insert)}}$.

$\mathcal{F}_{\text{MPC}(\text{insert})}$ in Game 7

Parameters: A next-step function NS

- Upon receiving (Run, Share$M$, Share$x$) from all parties:
  1. Ignore Share$M$, use $M \coloneqq M_{period}$ from the simulator state instead
  2. Reconstruct input x
  3. Initialize $\text{st}_{\text{NS}} \coloneqq (\text{start}, x)$, $data \coloneqq 0$ and $round \coloneqq 0$
  4. While $\text{st}_{\text{NS}} \neq (\text{stop}, z)$, do:
     a) Run $(\text{st}_{\text{NS}}, I) \leftarrow \text{NS}(\text{st}_{\text{NS}}, data)$
     b) If $I = (\text{read}, addr, \bot)$, set $data \coloneqq M[addr].val$
     c) Else if $I = (\text{write}, addr, val)$, set $data \coloneqq M[addr].val$ and update $M[addr].val = I.val$
     d) Update $round \coloneqq round + 1$
  5. Output shares Share_{NS}, Share$M$ and $round$ to all parties

$\mathcal{F}_{\text{MPC}(\text{insert})}$ in Game 8

Parameters: A next-step function NS

- Upon receiving (Run, Share$M$, Share$x$) from all parties:
  1. Ignore Share$M$, use $M \coloneqq M_{period}$ from the simulator state instead
  2. Reconstruct input x
  3. Initialize $\text{st}_{\text{NS}} \coloneqq (\text{start}, x)$, $data \coloneqq 0$ and $round \coloneqq 0$
  4. While $\text{st}_{\text{NS}} \neq (\text{stop}, z)$, do:
     a) Run $(\text{st}_{\text{NS}}, I) \leftarrow \text{NS}(\text{st}_{\text{NS}}, data)$
     b) If $I = (\text{read}, addr, \bot)$, set $data \coloneqq M[addr].val$
     c) Else if $I = (\text{write}, addr, val)$, set $data \coloneqq M[addr].val$ and update $M[addr].val = I.val$
     d) Update $round \coloneqq round + 1$
  5. Output shares Share_{NS}, Share0 and $round$ to all parties

Figure B.10.: Simulation of $\mathcal{F}_{\text{MPC}(\text{insert})}$, Part 2

Handling of (`createentry`, ...) in $\mathcal{F}^8$

- Upon receiving input (`createentry`, *ssid*, *period*, *entry*) from party P:
  1. Send (`in`, P, (`createentry`, *ssid*, *period*, *entry*)) to the simulator
- Upon receiving (`out`, P, msg) from the simulator:
  1. Send (msg) to party P

Handling of (`createentry`, ...) in $\mathcal{F}^9$

- Upon receiving input (`createentry`, *ssid*, *period*, *entry*) from party P:
  1. If pid_U is not in $\bigcup_{O \in \mathcal{O}} \mathsf{L}_{\text{users}}$, ignore this call
  2. Set $id^O_{\text{last}} \coloneqq id^O_{\text{last}} + 1$, $id \coloneqq id^O_{\text{last}}$
  3. Add $(id, pid_U, period, entry)$ to $\mathsf{L}_{\text{Pending}_O}$
  4. Send (`in`, P, (`createentry`, *ssid*, *period*, *entry*)) to the simulator
- Upon receiving (`out`, P, msg) from the simulator:
  1. Send (msg) to party P

Figure B.11.: First step in handling creation of log entries

Game 8. In this game, when simulating $\mathcal{F}_{\text{MPC}(\text{insert})}$, $\mathcal{S}^8$ outputs shares of an all-0 memory instead of Share$M$, see Fig. B.10.

Proof Sketch: Game 7 ≈ Game 8.
These games are identical from the environment's view since at most t operators are corrupted, and t out of the n shares reveal no information about M.

Game 9. In this game, $\mathcal{F}^9$ on input (`createentry`, *ssid*, *period*, *entry*) from U ignores the message if $pid_U \notin \bigcup_{O \in \mathcal{O}} \mathsf{L}_{\text{users}}$. Otherwise, it still sends the input to the simulator and delivers output as instructed by the simulator. Furthermore, $\mathcal{F}^9$ fills $\mathsf{L}_{\text{Pending}_O}$ the same way as $\mathcal{F}_{\text{POBA}}$. The concrete behavior of $\mathcal{F}^9$ is detailed in Fig. B.11.

Proof Sketch: Game 8 ≈ Game 9.
Storing internal data in $\mathcal{F}^9$ has no effect on the view of the environment. As such, the only observable change is $\mathcal{F}^9$ ignoring an input (`createentry`, ...). $\mathcal{F}^9$ only receives those inputs for honest users, and it only ignores the message if it did not receive a message (`ureg`, ...) before. Thus, whenever $\mathcal{F}^9$ ignores the message, the simulation of U by $\mathcal{S}^8$ would also abort since no values $(\text{sk}_U, \text{pk}_U, addr, \sigma)$ would have been stored for that user. Thus, from the environment's view, these games are identical.

Handling of `createentry` messages from corrupt users in $\mathcal{S}^9$

- When the adversary allows delivery of output $(\texttt{send}, ssid\|1, (entry_U, ct_U, \pi))$:
  1. Wait until receiving $(\texttt{in}, (\texttt{createentry}, ssid, period, entry_O), O)$ from $\mathcal{F}^9$
  2. If $entry_U \neq entry_O$, abort
  3. $h \coloneqq H(period, ct_U, entry_O)$, $stmt \coloneqq (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  4. If $\mathsf{Vf}(crs_{\mathsf{ZK}}, \mathcal{R}_{\mathsf{CE}}, stmt, \pi) \neq 1$, abort
  5. $id_{\text{last}} \coloneqq id_{\text{last}} + 1$
  6. Add $(id_{\text{last}}, period, ct_U, entry, \pi)$ to $\mathsf{L}_{\mathsf{Pending}}$
  7. $\sigma_{entry} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O, h)$
  8. Send $(\texttt{respond}, ssid\|1, (\sigma_{entry}, \mathsf{vk}_O))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ and $(\texttt{out}, O, (ssid, \texttt{ok}, id_{\text{last}}))$ to $\mathcal{F}^9$

Handling of `createentry` messages from corrupt users in $\mathcal{S}^{10}$

- When the adversary allows delivery of output $(\texttt{send}, ssid\|1, (entry_U, ct_U, \pi))$:
  1. Wait until receiving $(\texttt{in}, (\texttt{createentry}, ssid, period, entry_O), O)$ from $\mathcal{F}^{10}$
  2. If $entry_U \neq entry_O$, abort
  3. $h \coloneqq H(period, ct_U, entry_O)$, $stmt \coloneqq (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  4. If $\mathsf{Vf}(crs_{\mathsf{ZK}}, \mathcal{R}_{\mathsf{CE}}, stmt, \pi) \neq 1$, abort
  5. $id_{\text{last}} \coloneqq id_{\text{last}} + 1$
  6. Add $(id_{\text{last}}, period, ct_U, entry, \pi)$ to $\mathsf{L}_{\mathsf{Pending}}$
  7. $\sigma_{entry} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_O, h)$
  8. Extract $wit \coloneqq (\mathsf{pk}_U, \mathsf{sk}_U, addr, \sigma, r) \leftarrow \mathsf{ZK.Ext}(crs_{\mathsf{ZK}}, \mathcal{R}_{\mathsf{CE}}, stmt, \pi, td_{\mathsf{ZK}})$, abort if this fails
  9. Find $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$, abort if no such entry exists
  10. If U^* is honest, abort
  11. Send $(\texttt{createentry}, ssid, period, entry_U)$ to $\mathcal{F}_{\text{POBA}}$ on behalf of user U^*
  12. Send $(\texttt{respond}, ssid\|1, (\sigma_{entry}, \mathsf{vk}_O))$ to $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ and $(\texttt{out}, O, (ssid, \texttt{ok}, id_{\text{last}}))$ to $\mathcal{F}^{10}$

Figure B.12.: Calling $\mathcal{F}_{\text{POBA}}$ with $(\texttt{createentry}, \ldots)$ for corrupted users

Game 10. In this game, when receiving a message $(ssid, entry_U, ct_U, \pi)$ from a corrupted user, the simulator extracts the proper inputs and calls $\mathcal{F}^{10}$ in the name of the correct user, see Fig. B.12.

Proof Sketch: Game 9 ≈ Game 10.
The only change to the environment's view happens when the simulator aborts. This can happen because 1. extraction of the witness failed, 2. no matching entry in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$ exists, or 3. U^* is honest. The probability for (1) is bounded by the probability to break simulation-extractability of ZK. For (2), note that a valid signature σ on $(\mathsf{pk}_U, addr)$ has been extracted. That signature is generated during execution of the `ureg` task, and since Game 6 the simulator creates the appropriate entry in $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$ when generating this signature. Thus, no matching entry existing means σ was not created by the simulator, and thus the probability of (2) is bound by the probability to break TS-UF-1-security of TSIG (for the reduction, the simulator uses vk and the obtained sk_i for corrupted parties in the TS-UF-1 experiment during simulation of $\mathcal{F}^{\mathsf{TSIG}}_{\text{Keygen}}$ and replaces all instances of TSIG.PartSign of honest parties with calls to the signing oracle of the experiment). Lastly, for (3), extraction also provided the secret signing key of U^*. For a honest user, obtaining this is bounded by the probability to break EUF-CMA-security of SIG (for the reduction, the simulator uses vk of the EUF-CMA experiment. The user secret key is never actually used to generate a signature, and the simulator is already simulating all zero-knowledge proofs that would use the secret key. The extracted signing key can then be used to forge a valid signature). Thus, the distinguishing advantage is negligible.

Game 11. In this game, $\mathcal{F}^{11}$ on input $(\mathsf{insert}, ssid, period, id)$ from some O, ignores the call if no entry $(id, pid_U, period, entry)$ exists in $\mathsf{L}_{\mathsf{Pending}_O}$. Otherwise, it still sends the input to

Handling of (insert, ...) **in** $\mathcal{F}^{10}$

- Upon receiving input (insert, *ssid*, *period*, *id*) from party *P*:
  1. Send (in, *P*, (insert, *ssid*, *period*, *id*)) to the simulator
- Upon receiving (out, *P*, msg) from the simulator:
  1. Send (msg) to party *P*

Handling of (insert, ...) **in** $\mathcal{F}^{11}$

- Upon receiving input (insert, *ssid*, *period*, *id*) from party *P*:
  1. If no entry (*id*, pid_U, *period*, *entry*) exists in $\mathsf{L}_{\text{Pending}_O}$, ignore this call
  2. Otherwise, remove that entry and append (*entry*) to Log_U^{period}
  3. Send (in, *P*, (insert, *ssid*, *period*, *id*)) to the simulator.
- Upon receiving (out, *P*, msg) from the simulator:
  1. Send (msg) to party *P*

Figure B.13.: First step in handling creation of log entries

Handling of (proveentry, ...) **in** $\mathcal{F}^{11}$

- Upon receiving input (proveentry, *ssid*, *entry*, *period*, *O*) from user *U*:
  1. Send (in, *P*, (proveentry, *ssid*, *entry*, *period*, *O*)) to the simulator
- Upon receiving (out, *O*, (*ssid*, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. Send (*ssid*, proveentry, pid_U, *entry*, *period*) to *O*

Handling of (proveentry, ...) **in** $\mathcal{F}^{12}$

- Upon receiving input (proveentry, *ssid*, *entry*, *period*, *O*) from user *U*:
  1. If no entry (*id*, pid_U, *period*, *entry*) in $\mathsf{L}_{\text{Pending}_{O' \in O}}$ or (*entry*) in Log_U^{period} exists, ignore this input
  2. Otherwise, send (in, *P*, (proveentry, *ssid*, *entry*, *period*, *O*)) to the simulator
- Upon receiving (out, *O*, (*ssid*, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. Send (*ssid*, proveentry, pid_U, *entry*, *period*) to *O*

Figure B.14.: Handling of (proveentry) for honest *U* and *O* in $\mathcal{F}_{\text{POBA}}$

the simulator and delivers output as instructed by the simulator. Furthermore, $\mathcal{F}^{11}$ fills Log_U^{period} the same way as $\mathcal{F}_{\text{POBA}}$. The concrete behavior of $\mathcal{F}^{11}$ is detailed in Fig. B.13.

Proof Sketch: Game 10 ≈ Game 11.
Storing internal data in $\mathcal{F}^{11}$ has no effect on the view of the environment. As such, the only observable change is $\mathcal{F}^{11}$ ignoring a call instead of forwarding the input to the simulator. $\mathcal{F}^{11}$ ignores the call only if no matching entry (*id*, pid_U, *period*, *entry*) exists in $\mathsf{L}_{\text{Pending}_O}$. Since Game 10, such an entry is created whenever *O* successfully completes the createentry task. Thus, whenever $\mathcal{F}^{11}$ ignores the call, the simulation of the (semi-)honest *O* would also abort because no entry (*id*, *period*, ct_U, *entry*, π) exists in $\mathsf{L}_{\text{Pending}}$. Thus, from the environment's view, these games are identical.

Game 12. In this game, $\mathcal{F}^{12}$ ignores inputs (proveentry, *ssid*, *entry*, *period*, *O*) from some user *U* if no entry (*id*, pid_U, *period*, *entry*) in $\mathsf{L}_{\text{Pending}_{O' \in O}}$ or (*entry*) in Log_U^{period} exists. Otherwise, it still forwards these inputs to the simulator and forwards outputs from the simulator to *O*. For the concrete changes of this game, see Fig. B.14.

Proof Sketch: Game 11 ≈ Game 12.
$\mathcal{F}^{12}$ ignores inputs only if no matching entry exists. These entries were created since Game 10 resp. Game 11 whenever the user actually created the respective log entry. Thus, whenever $\mathcal{F}^{12}$ ignores an input, simulation of the honest user would also abort and thus

Handling of (proveentry, ...) in $\mathcal{F}^{12}$

- Upon receiving input (proveentry, *ssid*, *entry*, *period*, *O*) from user *U*:
  1. If no entry (*id*, pid_U, *period*, *entry*) in $\mathsf{L}_{\text{Pending}_{O' \in \mathcal{O}}}$ or (*entry*) in Log_U^{period} exists, ignore this input
  2. Otherwise, send (in, *P*, (proveentry, *ssid*, *entry*, *period*, *O*)) to the simulator
- Upon receiving (out, *O*, (*ssid*, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. Send (*ssid*, proveentry, pid_U, *entry*, *period*) to *O*

Handling of (proveentry, ...) in $\mathcal{F}^{13}$

- Upon receiving input (proveentry, *ssid*, *entry*, *period*, *O*) from user *U*:
  1. If no entry (*id*, pid_U, *period*, *entry*) in $\mathsf{L}_{\text{Pending}_{O' \in \mathcal{O}}}$ or (*entry*) in Log_U^{period} exists, ignore this input
  2. If *O* is honest:
     a) Send ((proveentry, *ssid*), *U*) to the adversary
     b) Output (*ssid*, proveentry, *U*, *entry*, *period*) to *O*
  3. Otherwise, send (in, *P*, (proveentry, *ssid*, *entry*, *period*, *O*)) to the simulator
- Upon receiving (out, *O*, (*ssid*, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. If *U* and *O* are honest, ignore this message
  2. Otherwise, send (*ssid*, proveentry, pid_U, *entry*, *period*) to *O*

Figure B.15.: Handling of (proveentry) for honest *U* and *O* in $\mathcal{F}_{\text{POBA}}$

Handling of (proveentry, ...) in $\mathcal{S}^{12}$

- Upon receiving (in, *U*, (proveentry, *ssid*, *entry*, *period*, *O*)) from $\mathcal{F}^{12}$:
  1. Run the protocol honestly for *U* and *O*
  2. When the protocol would generate output (*ssid*, proveentry, pid_U, *entry*, *period*) for *O*, send (out, *O*, (*ssid*, proveentry, pid_U, *entry*, *period*)) to $\mathcal{F}^{12}$

Handling of (proveentry, ...) in $\mathcal{S}^{13}$

- Upon receiving (in, *U*, (proveentry, *ssid*, *entry*, *period*, *O*)) from $\mathcal{F}^{13}$:
  1. Run the protocol honestly for *U* and *O*
  2. When the protocol would generate output (*ssid*, proveentry, pid_U, *entry*, *period*) for *O*, send (out, *O*, (*ssid*, proveentry, pid_U, *entry*, *period*)) to $\mathcal{F}^{13}$
- Upon receiving ((proveentry, *ssid*), *U*) from $\mathcal{F}^{13}$:
  1. Simply report a message of the appropriate length from *U* to *O*

Figure B.16.: Handling of (proveentry) for honest *U* and *O* in $\mathcal{S}$

not create any messages because no entry in $\mathsf{L}_{\text{Receipt}}$ exists. Therefore, these games are identical to the environment.

Game 13. In this game, $\mathcal{F}^{13}$ handles inputs (proveentry, *ssid*, *entry*, *period*, *O*) from some user *U* the same way as $\mathcal{F}_{\text{POBA}}$ does if both *U* and *O* are honest, and $\mathcal{S}^{13}$ handles simulation of them the same way as $\mathcal{S}$ does. For the concrete changes of this game, see Figs. B.15 and B.16.

Proof Sketch: Game 12 ≈ Game 13.
This is indistinguishable for the environment as running the real protocol reveals nothing besides the length of the message from *U* to *O*.

Game 14. In this game, $\mathcal{F}^{14}$ handles inputs (proveentry, *ssid*, *entry*, *period*, *O*) for honest users same way as $\mathcal{F}_{\text{POBA}}$ does, and $\mathcal{S}^{14}$ handles simulation of them the same way as $\mathcal{S}$ does for honest users. For corrupted users, $\mathcal{S}^{14}$ still handles them the same way as $\mathcal{S}^{13}$

Handling of (proveentry, ...) in $\mathcal{F}^{13}$

- Upon receiving input (proveentry, *ssid*, *entry*, *period*, *O*) from user *U*:
  1. If no entry (*id*, pid_U, *period*, *entry*) in $\mathsf{L}_{\mathsf{Pending}_{O' \in O}}$ or (*entry*) in Log_U^{period} exists, ignore this input
  2. If *O* is honest:
     a) Send ((proveentry, *ssid*), *U*) to the adversary
     b) Output (*ssid*, proveentry, pid_U, *entry*, *period*) to *O*
  3. Otherwise, send (in, *P*, (proveentry, *ssid*, *entry*, *period*, *O*)) to the simulator
- Upon receiving (out, *O*, (*ssid*, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. If *U* and *O* are honest, ignore this message
  2. Otherwise, send (*ssid*, proveentry, pid_U, *entry*, *period*) to *O*

Handling of (proveentry, ...) in $\mathcal{F}^{14}$

- Upon receiving input (proveentry, *ssid*, *entry*, *period*, *O*) from some user *U*:
  1. Check if an entry (*id*, pid_U, *period*, *entry*) exists in $\mathsf{L}_{\mathsf{Pending}_{O' \in O}}$ or an entry (*entry*) exists in Log_U^{period}, otherwise ignore this input
  2. If *O* is corrupted, send ((proveentry, *ssid*, *entry*, *period*, *O*), *U*) to the adversary
     Otherwise, send ((proveentry, *ssid*), *U*) to the adversary.
  3. Output (*ssid*, proveentry, *U*, *entry*, *period*) to *O*
- Upon receiving (out, *O*, (*ssid*, proveentry, pid_U, *entry*, *period*)) from the simulator:
  1. If *U* is honest, ignore this message
  2. Otherwise, send (*ssid*, proveentry, pid_U, *entry*, *period*) to *O*

Figure B.17.: Handling of (proveentry) in $\mathcal{F}_{\text{POBA}}$

Handling of (proveentry, ...) in $\mathcal{S}^{13}$

- Upon receiving ((proveentry, *ssid*), *U*) from $\mathcal{F}^{13}$:
  1. Simply report a message of the appropriate length from *U* to *O*
- Upon receiving (in, *U*, (proveentry, *ssid*, *entry*, *period*, *O*)) from $\mathcal{F}^{13}$:
  1. Run the protocol honestly for *U* and *O*
  2. When the protocol would generate output (*ssid*, proveentry, pid_U, *entry*, *period*) for *O*, send (out, *O*, (*ssid*, proveentry, pid_U, *entry*, *period*)) to $\mathcal{F}^{13}$

Handling of (proveentry, ...) in $\mathcal{S}^{14}$

- Upon receiving ((proveentry, *ssid*), *U*) from $\mathcal{F}^{14}$:
  1. Simply report a message of the appropriate length from *U* to *O*
- Upon receiving ((proveentry, *ssid*, *entry*, *period*, *O*), *U*) from $\mathcal{F}^{14}$:
  1. Retrieve $(entry, ct_U, r, \sigma_{entry}, \mathsf{vk}_O)$ from $\mathsf{L}^{\mathsf{U}}_{\mathsf{Receipt}}$
  2. Retrieve $(\mathsf{pk}_U, \perp, addr, U)$ from $\mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  3. Set $stmt \coloneqq (\mathsf{pk}_U, addr, ct_U)$
  4. Set $\pi \leftarrow \mathsf{ZK.Sim}(crs_{\mathsf{ZK}}, \mathcal{R}_{\mathsf{PE}}, stmt, td_{\mathsf{ZK}})$
  5. Report output $(ssid\|1, pid_U, (period, entry, \mathsf{pk}_U, addr, \sigma, ct_U, \sigma_{entry}, \mathsf{vk}_O, \pi))$ to *O* from $\mathcal{F}_{\text{SMT}}$
- When the adversary allows delivery of that output:
  1. Execute the protocol for *O* honestly, sending the party's state to the adversary after each activation and waiting for continue from the adversary before sending out any messages.
  2. When the adversary allows *O* to generate output, allow $\mathcal{F}^{14}$ to deliver output to *O*

Figure B.18.: Handling of (proveentry) for honest *U* in $\mathcal{S}$

and $\mathcal{F}^{14}$ still forwards output to *O*. For the concrete changes of this game, see Figs. B.17 and B.18.

Proof Sketch: Game 13 ≈ Game 14.
The only visible difference between these games for the environment is how π in the message to the semi-honest operator is generated. Thus, the probability of distinguishing these games is bounded by the probability to break zero-knowledge of ZK and therefore the games are indistinguishable.

Handling of (proveentry, ...) in $\mathcal{F}^{14}$

- When the adversary allows delivery of output $(ssid \| 1, pid_U, (period, entry, \mathsf{pk}_U, addr, \sigma, ct_U, \sigma_{entry}, \mathsf{vk}_O, \pi))$ from $\mathcal{F}_{\text{SMT}}$:
  1. Check if an entry $(id, pid_U, period, entry)$ exists in $\mathsf{L}_{\mathsf{Pending}_{O' \in \mathcal{O}}}$ or an entry $(entry)$ exists in Log_U^{period}, otherwise ignore this input
  2. If O is corrupted, send $((\texttt{proveentry}, ssid, entry, period, O), U)$ to the adversary, otherwise send $((\texttt{proveentry}, ssid), U)$ to the adversary
  3. Output $(ssid, \texttt{proveentry}, U, entry, period)$ to O
- Upon receiving $(\texttt{out}, O, (ssid, \texttt{proveentry}, pid_U, entry, period))$ from the simulator:
  1. If U is honest, ignore this message
  2. Otherwise, send $(ssid, \texttt{proveentry}, pid_U, entry, period)$ to O

Handling of (proveentry, ...) in $\mathcal{F}^{15}$

- Upon receiving input $(\texttt{proveentry}, ssid, entry, period, O)$ from some user U:
  1. Check if an entry $(id, pid_U, period, entry)$ exists in $\mathsf{L}_{\mathsf{Pending}_{O' \in \mathcal{O}}}$ or an entry $(entry)$ exists in Log_U^{period}, otherwise ignore this input
  2. If O is corrupted, send $((\texttt{proveentry}, ssid, entry, period, O), U)$ to the adversary, otherwise send $((\texttt{proveentry}, ssid), U)$ to the adversary
  3. Output $(ssid, \texttt{proveentry}, U, entry, period)$ to O
- Upon receiving $(\texttt{out}, O, (ssid, \texttt{proveentry}, pid_U, entry, period))$ from the simulator:
  1. Ignore this message

Figure B.19.: Handling of (proveentry) in $\mathcal{F}_{\text{POBA}}$

Handling of (proveentry, ...) in $\mathcal{S}^{14}$

- Upon receiving $((\texttt{proveentry}, ssid), U)$ from $\mathcal{F}^{15}$:
  1. Simply report a message of the appropriate length from U to O
- Upon receiving $((\texttt{proveentry}, ssid, entry, period, O), U)$ from $\mathcal{F}^{15}$:
  1. [...]
- Upon receiving $(ssid, period, entry, \mathsf{pk}_U, addr, \sigma, ct_U, \sigma_{entry}, \mathsf{vk}_O, \pi)$ from a corrupted U:
  1. Follow the protocol of O honestly, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages if O is semi-honest
  2. When O generates output, send the appropriate $(\texttt{out}, O, \ldots)$ message to $\mathcal{F}^{14}$

Handling of (proveentry, ...) in $\mathcal{S}^{15}$

- Upon receiving $((\texttt{proveentry}, ssid), U)$ from $\mathcal{F}^{15}$:
  1. Simply report a message of the appropriate length from U to O
- Upon receiving $((\texttt{proveentry}, ssid, entry, period, O), U)$ from $\mathcal{F}^{15}$:
  1. [...]
- Upon receiving $(ssid, period, entry, \mathsf{pk}_U, addr, \sigma, ct_U, \sigma_{entry}, \mathsf{vk}_O, \pi)$ from a corrupted U:
  1. Follow the steps of the honest protocol, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages if O is semi-honest
  2. Before asking the adversary for permission to deliver output:
     a) Set $stmt \coloneqq (\mathsf{pk}_U, addr, ct_U)$
     b) Get $wit \coloneqq (r, \mathsf{sk}_U) \leftarrow \text{ZK.Ext}(crs_{\text{ZK}}, \mathcal{R}_{\mathsf{PE}}, stmt, \pi, td_{\text{ZK}})$, abort if this fails
     c) Find the entry $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}_{\text{Keys}}^{\mathcal{S}}$, abort if no matching entry exists
     d) If U^* is honest, abort
     e) Send message $(\texttt{proveentry}, entry, period, O, ssid)$ to $\mathcal{F}^{15}$ on behalf of U^*
  3. When the adversary allows O to deliver output, allow $\mathcal{F}^{15}$ to deliver output to O

Figure B.20.: Handling of (proveentry) for honest U in $\mathcal{S}$

Game 15. In this game, $\mathcal{F}^{15}$ handles inputs $(\texttt{proveentry}, ssid, entry, period, O)$ same way as $\mathcal{F}_{\text{POBA}}$ does, and $\mathcal{S}^{15}$ handles simulation of them the same way as $\mathcal{S}$ does for both honest and corrupted users. For the concrete changes of this game, see Figs. B.19 and B.20.

Handling of `analytics` in $\mathcal{F}^{15}$

- Upon receiving input (`analytics`, *ssid*, *func*, $\{period_i\}$, $\{U_i\}$, aux_O) from O:
  1. Send (`in`, O, (`analytics`, *ssid*, *func*, $\{period_i\}$, $\{U_i\}$)) to the simulator
- Upon receiving (`out`, O, (*ssid*, $\texttt{result}_O$)) from the simulator:
  1. Send (*ssid*, $\texttt{result}_O$) to O

Handling of `analytics` in $\mathcal{F}^{16}$

- Upon receiving input (`analytics`, *ssid*, *func*, $\{period_i\}$, $\{U_i\}$, aux_O) from O:
  1. If not all parties provided the same *func*, $\{period_i\}$ and $\{U_i\}$ as input, ignore this call
  2. Set Logs $\coloneqq \{\text{Log}_U^p\}_{U\in\{U_i\},p\in\{period_i\}}$
     Evaluate $\{\texttt{result}_O\}_{O\in\mathcal{O}} \leftarrow$ *func*(Logs, $\{aux_O\}_{O\in\mathcal{O}}$), setting *sequence* $\coloneqq (p_1, p_2, \ldots, p_n)$ with an entry $p_j \coloneqq period_j$ for each time *func* reads an entry from $\text{Log}_U^{period_j}$
  3. If at least one operator is semi-honest, send (`analytics`, *ssid*, *sequence*) to the adversary
  4. For each operator O, generate delayed private output (*ssid*, $\texttt{result}_O$)

Figure B.21.: Handling of analytics in $\mathcal{F}_{\text{POBA}}$

Proof Sketch: Game 14 ≈ Game 15.
Let us first consider when the simulator aborts. This is when 1. extraction of the witness fails, 2. no matching entry in $\text{L}^{\mathcal{S}}_{\text{Keys}}$ exists, or 3. U^* is honest. The probability of (1) is bounded by the probability to break simulation-extractability of ZK. The message contains a valid signature σ on (pk_U, *addr*). Whenever such a signature is created, the appropriate entry in $\text{L}^{\mathcal{S}}_{\text{Keys}}$ is also generated, thus the probability for (2) is bounded by the probability to break TS-UF-1-security of TSIG. Regarding (3), extraction provided sk_U, obtaining of which for an honest user is as hard as breaking EUF-CMA-security of SIG. Therefore, the probability that $\mathcal{S}^{15}$ aborts is negligible. Note that when $\mathcal{S}^{15}$ sends (`proveentry`, ...) to $\mathcal{F}^{15}$, simulation of O outputs (*ssid*, `proveentry`, ...). We thus need to ensure that $\mathcal{F}^{15}$ also delivers that output to O. This is the case because if $\mathcal{S}^{15}$ did not abort, an appropriate call of (`createentry`, ...) has been made.

Game 16. In this game, $\mathcal{F}^{16}$ handles inputs (`analytics`, ...) the same way as $\mathcal{F}_{\text{POBA}}$ and $\mathcal{S}^{16}$ handles simulation of those inputs and $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ the same way as $\mathcal{S}$. For the concrete changes, see Figs. B.21 and B.22.

Proof Sketch: Game 15 ≈ Game 16.
If all operators are honest, the only thing visible to the environment are the lengths of the messages required to reveal shares of the outputs. Since shares have a fixed length, simulation of these is trivial and perfect. With at least one semi-honest operator, $\mathcal{F}^{16}$ leaks *sequence* required to simulate $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ as well as the correct output for each semi-honest operator. With these information, simulation of $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ is perfect. The only other messages visible to semi-honest operators and thus the environment are those required to reveal shares of their output, and since $\mathcal{S}^{16}$ was using the correct outputs when generating these shares, these are also identically distributed.

Game 17. In this game, $\mathcal{F}^{17}$ handles inputs (insert, ...) the same way as $\mathcal{F}_{\text{POBA}}$ and $\mathcal{S}^{17}$ handles simulation of them the same way as $\mathcal{S}$. For the concrete changes, see Figs. B.23 and B.24.

Handling of `analytics` in $\mathcal{S}^{15}$

- Upon receiving (`in`, O, (`analytics`, *ssid*, ...)) from $\mathcal{F}^{15}$
  1. Run the protocol honestly from O
  2. When the protocol would generate output (*ssid*, $\texttt{result}_O$) for O send (`out`, O, (*ssid*, $\texttt{result}_O$)) to $\mathcal{F}^{15}$

Handling of `analytics` in $\mathcal{S}^{16}$

- If all operators are honest, simply report messages for revealing output shares of the appropriate lengths.
- Otherwise (at least one operator is semi-honest):
- Upon receiving ((`analytics`, *ssid*, *func*, $\{period_i\}$, $\{U_i\}$), O') from $\mathcal{F}_{\text{POBA}}$ (i.e., O' is semi-honest):
  1. Simulate O' honestly, sending the state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages. When the adversary allows generation of output, allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O'
- Upon receiving ((`analytics`, *ssid*), O) from $\mathcal{F}_{\text{POBA}}$ (i.e., O is honest):
  1. Do nothing
- Wait until receiving (`analytics`, *ssid*, *sequence*) and outputs (*ssid*, $\texttt{result}_{O'}$) for each semi-honest operator O', then simulate $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ as follows:
  1. For each honest O^*, set $\texttt{result}_{O^*} \coloneqq 0$
  2. Set $\texttt{result} \coloneqq \bigcup \texttt{result}_O$ (this is 0 for honest operators, and the output of $\mathcal{F}_{\text{POBA}}$ for semi-honest operators)
  3. Output shares $\text{Share}(\text{stop}, \texttt{result})$, $\{\text{Share}M_i\}$, *round*, *sequence* to all parties
- When the simulation of $\mathcal{F}^{\text{ext}}_{\text{MPC}(func)}$ has delivered output to honest O:
  1. For each semi-honest O', reveal $\text{Share}\texttt{result}_{O'}$ to O'
  2. For each honest O^*, report a message of the correct length from O to O^*
- When the adversary allowed delivery of all share reveals towards an honest operator O:
  1. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output to O

Figure B.22.: Handling of analytics in $\mathcal{S}$

Handling of (insert, ...) in $\mathcal{F}^{16}$

- Upon receiving input (insert, *ssid*, *period*, *id*) from O:
  1. If no entry (*id*, pid_U, *period*, *entry*) exists in $\mathsf{L}_{\text{Pending}_O}$, ignore this call
  2. Otherwise, remove that entry and append (*entry*) to Log_U^{period}
  3. Send (`in`, P, (insert, *ssid*, *period*, *id*)) to the simulator
- Upon receiving (`out`, P, msg) from the simulator:
  1. Send (msg) to party P

Handling of (insert, ...) in $\mathcal{F}^{17}$

- Upon receiving input (insert, *ssid*, *period*, *id*) from O and (insert, *ssid*) from all other O':
  1. If at least one O' is corrupted, send ((insert, *ssid*, *period*, *id*), O) to the adversary
  2. If no entry (*id*, pid_U, *period*, *entry*) exists in $\mathsf{L}_{\text{Pending}_O}$, ignore this call
  3. Otherwise, remove that entry and append (*entry*) to Log_U^{period}
  4. Generate delayed private output (*ssid*, ok) for each O

Figure B.23.: Handling insertion of log entries in $\mathcal{F}_{\text{POBA}}$

Proof Sketch: Game 16 ≈ Game 17.
Simulation of $\mathcal{F}_{\text{MPC(insert)}}$ is the same in both games as insert has no output and the number of memory accesses is constant, independent of $addr^*$. The simulator now does not correctly update M anymore, but since Game 16 this is not used in a way visible to the environment anymore, so this change is indistinguishable for the environment. Lastly, the messages sent to semi-honest parties are distributed the same in both games since ct_U,*entry* and π are the same in both games. Thus, these games are perfectly indistinguishable to the environment.

Handling of (insert, ...) **in** $\mathcal{S}^{16}$

- Upon receiving (in, O, (insert, ...)) from $\mathcal{F}^{16}$
  1. Run the protocol honestly for O
  2. When the protocol would generate output (*entry*) for O, send (out, O, *entry*) to $\mathcal{F}^{16}$

Handling of (insert, ...) **in** $\mathcal{S}^{17}$

- Upon receiving ((insert, *ssid*, *period*, *id*), O) from $\mathcal{F}^{17}$ (i.e., O is semi-honest):
  1. Honestly execute the protocol for O, sending the party's state to the adversary after each activation and waiting for continue from the adversary before sending out any messages.
- Upon receiving ((insert, *ssid*), O) and ((insert, *ssid*, *period*, *entry**), O) from $\mathcal{F}^{17}$ (i.e., O is honest):
  1. Find (id, *period*, ct_U, *entry*, π) $\in \mathsf{L}^{\mathsf{O}}_{\mathsf{Pending}}$ and ignore this message if no entry with (id, *period*) exists
  2. Send (*ssid*, ct_U, *entry*, π) to all O'
  3. Honestly run the checks from the real protocol
  4. Treat O as having provided input to $\mathcal{F}_{\text{MPC(insert)}}$
- Upon receiving ((insert, *ssid*), O') from $\mathcal{F}^{17}$ for honest O':
  1. Wait until the adversary allowed delivery of the message (ct_U, *entry*, π) from O
  2. Honestly run the checks from the real protocol
  3. Treat O' as having provided input to $\mathcal{F}_{\text{MPC(insert)}}$

Simulate $\mathcal{F}_{\text{MPC(insert)}}$ as follows:

- Upon receiving (Run, Share$M$, Share$x$) from all semi-honest operators and having treated all honest operators as having provided input to $\mathcal{F}_{\text{MPC(insert)}}$:
  1. Choose a random $addr^* \in \mathsf{L}^{\mathcal{S}}_{\mathsf{Keys}}$
  2. Ignore Share$M$ and Share$x$, use M and partial decryptions of TPKE.Encrypt(ek, $addr^*$) instead
  3. Follow the description of insert honestly
  4. When delivering output to semi-honest operators, instead of Share$M$ use Share0, where 0 is a memory the same size as M but consisting only of 0-entries
- When the adversary allows delivery of output from $\mathcal{F}_{\text{MPC(insert)}}$ to an operator O
  1. Allow $\mathcal{F}^{17}$ to deliver output to O

Figure B.24.: Handling insertion of log entries in $\mathcal{S}$

Game 18. In this game, when both user and operator are honest while calling createentry, $\mathcal{F}^{18}$ no longer sends their inputs to the simulator. $\mathcal{S}^{18}$ simulates this case by simply reporting messages of appropriate size between the honest parties. Since this results in $\mathcal{S}^{18}$ no longer creating entries (id, *period*, ct_U, *entry*, π) in $\mathsf{L}^{\mathsf{O}}_{\mathsf{Pending}}$, it now uses the leak of (*period*, *entry*) provided by $\mathcal{F}^{18}$ to simulate InsertEntry for such entries. For the detailed changes to $\mathcal{S}^{18}$ regarding InsertEntry, see Fig. B.25.

Proof Sketch: Game 17 ≈ Game 18.
First, let us observe the changes to InsertEntry: The probability of distinguishing the change of ct_U during insert is bounded by the probability to break IND-CPA-security of TPKE, and of π by the probability to break zero-knowledge of ZK. The message is still the same, as the one leaked by $\mathcal{F}^{18}$ is the same that was sent by $\mathcal{F}^{17}$ during CreateEntry.

Game 19. We now further change createentry: In this game, when the user is honest but the operator is only semi-honest, $\mathcal{F}^{19}$ no longer forwards the user's input or output from the simulator. Since the operator is semi-honest, $\mathcal{S}^{19}$ still gets their input to simulate. See Fig. B.26 for the detailed changes.

Handling of (insert, ...) **in** $\mathcal{S}^{17}$

- Upon receiving $((\text{insert}, ssid), O)$ and $((\text{insert}, ssid, period, entry^*), O)$ from $\mathcal{F}^{18}$ (i.e., O is honest):
  1. Find $(id, period, ct_U, entry, \pi) \in \mathsf{L}^{O}_{\text{Pending}}$ and ignore this message if no entry with $(id, period)$ exists
  2. Send $(ssid, ct_U, entry, \pi)$ to all O'
  3. Honestly run the checks from the real protocol
  4. Treat O as having provided input to $\mathcal{F}_{\text{MPC}(\text{insert})}$

Handling of (insert, ...) **in** $\mathcal{S}^{18}$

- Upon receiving $((\text{insert}, ssid), O)$ and $((\text{insert}, ssid, period, entry^*), O)$ from $\mathcal{F}^{18}$ (i.e., O is honest):
  1. Create $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, 0)$
  2. Set $h \coloneqq H(period, ct_U, entry^*)$
  3. Set $stmt \coloneqq (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  4. Set $\pi \leftarrow \mathsf{ZK.Sim}(crs_{\mathsf{ZK}}, \mathcal{R}_{\mathsf{CE}}, stmt, td_{\mathsf{ZK}})$
  5. Send $(ssid, ct_U, entry^*, \pi)$ to all O'
  6. Treat O as having provided input to $\mathcal{F}_{\text{MPC}(\text{insert})}$

Figure B.25.: Handling insertion of log entries in $\mathcal{S}$

Handling of `createentry` **in** $\mathcal{S}^{18}$

- Upon receiving $(\texttt{in}, O, (\texttt{createentry}, ssid, period, entry_O))$ and $(\texttt{in}, U, (\texttt{createentry}, ssid, period, entry_U))$:
  1. Run the protocol honestly
- When the adversary allows delivery of the output:
  1. Forward that output to $\mathcal{F}^{18}$

Handling of `createentry` **in** $\mathcal{S}^{19}$

- Honestly run the protocol for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages from it.
- Upon receiving $((\texttt{createentry}, ssid), \texttt{user})$ and $((\texttt{createentry}, ssid, period, entry), O)$ from $\mathcal{F}_{\text{POBA}}$:
  1. Set $addr \coloneqq 0$
  2. Generate $ct_U \leftarrow \mathsf{TPKE.Encrypt}(\mathsf{ek}, addr; r)$
  3. Set $h \coloneqq H(period, ct_U, entry)$
  4. Set $stmt \coloneqq (ct_U, \mathsf{ek}, crs_{\mathsf{TSIG}}, \mathsf{vk}, h)$
  5. Set $\pi \leftarrow \mathsf{ZK.Sim}(crs_{\mathsf{ZK}}, \mathcal{R}_{\mathsf{CE}}, stmt, td_{\mathsf{ZK}})$
  6. Report that $\mathcal{F}^{\text{os-auth}}_{\text{SMT}}$ wants to generate output $(\texttt{send}, ssid\|1, (ct_U, \pi))$ to O
- When the adversary allows delivery of output $(\texttt{respond}, ssid\|1, (\sigma_{entry}, \mathsf{vk}_O))$:
  1. Store $(entry, ct_U, r, \sigma_{entry}, \mathsf{vk}_O)$ in $\mathsf{L}^{U}_{\text{Receipt}}$
  2. Allow $\mathcal{F}_{\text{POBA}}$ to deliver output

Figure B.26.: Handling creation of log entries in $\mathcal{S}$

Proof Sketch: Game 18 ≈ Game 19.
Again, these games are indistinguishable because the only changes are the generation of ct_U and π. Thus, the distinguishing advantage is bounded by the probability to break either IND-CPA of TPKE or zero-knowledge of ZK.

Game 20. Another change to `createentry`: In this game, when the user is corrupt and the operator is semi-honest, $\mathcal{F}^{20}$ handles this task the same way as $\mathcal{F}_{\text{POBA}}$ does, and $\mathcal{S}^{20}$ the same way as $\mathcal{S}$ does. Since the operator is semi-honest, $\mathcal{S}^{20}$ still gets the operator's input to simulate. See Fig. B.27 for the detailed changes.

Proof Sketch: Game 19 ≈ Game 20.
The simulation of the operator stays the same. Thus, these games differ either because the simulator aborts or because $\mathcal{F}^{20}$ delivers different output than the real protocol. The probability for the first abort is bound by the probability of breaking simulation-extractability of ZK. The probability of the second abort is bounded by the probability of breaking TS-UF-01-security of TSIG, because extraction provided a valid signature, and whenever that signature is generated, the simulator creates the respective entry in $\mathsf{L}^{\mathcal{S}}_{\text{Keys}}$.

Handling of `createentry` in $\mathcal{S}^{19}$

- Upon receiving $(\texttt{in}, O, (\texttt{createentry}, ssid, period, entry_O))$ and outputs from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$:
  1. Run the protocol honestly
- When the adversary allows delivery of output for the operator:
  1. Forward that output to $\mathcal{F}^{19}$

Handling of `createentry` in $\mathcal{S}^{20}$

- The simulator executes the protocol honestly for O, sending the party's state to the adversary after each activation and waiting for `continue` from the adversary before sending out any messages.
- Additionally, the simulator does the following:
  - When the adversary allows delivery of output $(\texttt{send}, ssid\|1, (entry_U, ct_U, \pi))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$, if the honest protocol did not abort as a result of this message:
    1. Get $wit := (\mathsf{pk}_U, \mathsf{sk}_U, addr, \sigma, r) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}}, \mathcal{R}_{\text{CE}}, stmt, \pi, td_{\text{ZK}})$
    2. Find $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}_{\text{Keys}}^{\mathcal{S}}$, abort if no such entry exists
    3. If U^* is honest, abort
    4. Send $(\texttt{createentry}, ssid, period, entry_U)$ to $\mathcal{F}_{\text{POBA}}$ on behalf of user U^*

Figure B.27.: Handling creation of log entries in $\mathcal{S}$

Handling of `createentry` in $\mathcal{S}^{20}$

- Upon receiving $(\texttt{in}, O, (\texttt{createentry}, ssid, period, entry_O))$ and outputs from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$:
  1. Run the protocol honestly
- When the adversary allows delivery of output for the operator:
  1. Forward that output to $\mathcal{F}^{20}$

Handling of `createentry` in $\mathcal{S}^{21}$

- When the adversary allows delivery of output $(\texttt{send}, ssid\|1, (entry_U, ct_U, \pi))$ from $\mathcal{F}_{\text{SMT}}^{\text{os-auth}}$ and after receiving $((\texttt{createentry}, ssid), O)$ from $\mathcal{F}^{21}$:
  1. $h := H(period, ct_U, entry_U)$, $stmt := (ct_U, \mathsf{ek}, crs_{\text{TSIG}}, \mathsf{vk}, h)$
  2. If $\mathsf{Vf}(crs_{\text{ZK}}, \mathcal{R}_{\text{CE}}, stmt, \pi) \neq 1$, ignore this message
  3. Otherwise, get $wit := (\mathsf{pk}_U, \mathsf{sk}_U, addr, \sigma, r) \leftarrow \mathsf{ZK.Ext}(crs_{\text{ZK}}, \mathcal{R}_{\text{CE}}, stmt, \pi, td_{\text{ZK}})$, abort if this fails
  4. Find $(\mathsf{pk}_U, \mathsf{sk}_U, addr, U^*)$ in $\mathsf{L}_{\text{Keys}}^{\mathcal{S}}$, abort if no such entry exists
  5. If U^* is honest, abort
  6. Send $(\texttt{createentry}, ssid, period, entry_U)$ to $\mathcal{F}_{\text{POBA}}$ on behalf of user U^*

Figure B.28.: Handling creation of log entries in $\mathcal{S}$

The probability for the third abort is bounded by the EUF-CMA-security of SIG, since extraction provided the secret key of the user, which can be used to win the EUF-CMA game. Note that this is possible because by this point, the secret key of honest users is not used anywhere by the simulator anymore. If $\mathcal{S}^{20}$ does not abort, then $\mathcal{F}^{20}$ delivers the same output as the real protocol: It does not ignore the call because U^* is corrupted, has called `ureg` previously and the messages match, and *id* is calculated the same way by $\mathcal{F}_{\text{POBA}}$ and the protocol.

Game 21. We again change `createentry`: In this game, when the user is corrupt and the operator is honest, $\mathcal{F}^{21}$ handles this task the same way as $\mathcal{F}_{\text{POBA}}$ does, and $\mathcal{S}^{21}$ the same way as $\mathcal{S}$ does. This means that in this game, $\mathcal{F}^{21}$ handles `createentry` as a whole the same as $\mathcal{F}_{\text{POBA}}$ does, and $\mathcal{S}^{21}$ handles it the same as $\mathcal{S}$ does. See Fig. B.28 for the detailed changes.

Proof Sketch: Game 20 ≈ Game 21.
These games are indistinguishable for the same reason as the two previous games. The simulator does not abort because of simulation-extractability of ZK, TS-UF-01-security of TSIG and EUF-CMA-security of SIG. When it does not abort, the output of the ideal functionality matches that of the real protocol.

Note that $\mathcal{S}^{21}$ equals the final simulator $\mathcal{S}$. Since we showed that Game i is indistinguishable from Game $i + 1$ for $i \in \{0, \ldots, 20\}$, it follows that Game 0 (the real protocol) is indistinguishable from Game 21 (ideal execution) and thus Theorem B.2.1 follows. □