On (not) Trusting Trusted Hardware

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte
Dissertation

von

Felix Dörre
aus Stuttgart

Tag der mündlichen Prüfung: 10. November 2025
1. Referent: Prof. Dr. Jörn Müller-Quade
2. Referent: Prof. Dr. Frederik Armknecht

Abstract

Sharing data between entities is ubiquitous, often for data processing on behalf of a data owner, or to enable a joint computation between the entities. Over the years several techniques have been found to ensure that other entities process the data as agreed upon. Secure computations can ensure that all entities cannot learn additional information about the data or manipulate the output of the computation. This building block can even enable computations where legal requirements prohibit merging different data sources (e.g. calculations on medical data).

To realize secure computation there are two main approaches: Secure multi-party computation (MPC) uses cryptographic techniques to hide data from participants and ensure that they follow the agreed upon computation. In general, MPC can be realized for arbitrary functions, for example with secret sharing, garbled circuits or homomorphic encryption. In many real-world scenarios however, implementations often need to be tailored for a specific and simple computation to achieve acceptable performance. Enclaves (often sold under the terms "Trusted Computing" and "Confidential Computing") claim to realize secure computation with only minimal performance penalty, compared to programs running on regular computer hardware. Depending on the concrete realization, enclaves claim isolation from other software running on the same system or even specific physical attacks. In the end, enclaves provide some form of "attestation", to allow others to verify that the computation has taken place in an enclave of a specific manufacturer. The downside of this approach is, that for the integrity and confidentiality of the computation, the user needs to assume that all enclaves produced by the manufacturer cannot be successfully attacked. One insecure enclave is sufficient for an attacker to forge attestations and attack the computation. In contrast, MPC only requires trust in one's own hardware and the security of the cryptographic protocol.

This dissertation shows how to combine the advantages of both MPC and enclaves, leading to protocols with better performance and/or security than pure MPC protocols, while only requiring little trust in the security of the enclaves used. This shows that trust is not a binary choice, but depending on trust given the resulting protocol can have different characteristics for security and performance. On top of that, this dissertation shows that complete trust in one's own hardware is not necessarily a prerequisite for running e.g. MPC protocols, so a protocol with even less trust than usually assumed can be realized.

In particular we construct the fastest to date protocol to implement private set intersection using enclaves with reduced trust assumptions.Apart from its many applications, private set itersection is an interesting problem to study as it has many specialized and highly performant MPC protocols. Our construction brings the performance cost even further down by using enclaves, while only assuming little trust in the enclaves. For another application, federated decision tree training, we construct a performant specialized MPC protocol with some leakage. Additionally, we show how it can be enhanced with enclaves, leading to different security levels depending on the security that is assumed from the enclave. If the enclaves are fully secure, the enhanced protocol has no leakage, if the enclaves have memory access side channels the protocol has an intermediate leakage, and if the enclaves have all possible side-channels, the

protocol has the same leakage as it would have without enclaves. What is more, the thesis shows that even the generally assumed trust in one's own hardware for MPC does not need to be taken for granted. We show how to realize secure computation with less trust assumptions than generally accepted for MPC at a performance cost. To achieve this result, we take an arbitrary execution of a whole virtual machine and trace non-deterministic behavior. This execution is then verified on a second machine (possibly from a different vendor) achieving security when one of the machines behaves honestly.

Zusammenfassung

Überall werden Daten geteilt. Oft für Berechnungen im Auftrag des Besitzers oder für gemeinsame Berechnungen zwischen mehreren Entitäten. Über die Jahre wurden verschiedene Techniken gefunden, die sicherstellen, dass andere Teilnehmer Daten nur so verarbeiten, wie vereinbart. Sichere Berechnungen können sicherstellen, dass alle Teilnehmer keine zusätzliche Information über die Daten lernen oder die Ausgabe der Berechnung verändern. Dieser Baustein kann sogar dann Berechnung ermöglichen, wenn rechtliche Anforderungen das Zusammenführen von Daten aus mehreren Quellen verbieten (z. B. bei Berechnungen auf Medizindaten).

Es existieren zwei Ansätze um sichere Berechnungen zu realisieren: Sichere Mehrparteienberechnung (MPC) benutzt kryptografische Techniken um Daten vor Teilnehmern zu verstecken und sicherzustellen, dass alle der vereinbarten Berechnung folgen. Im Allgemeinen lässt sich MPC für beliebige Funktionen realisieren, z.B. durch den Einsatz von Secret Sharing, Garbled Circuits oder homomorpher Verschlüsselung. In vielen realen Szenarien müssen MPC-Implementierungen aber auf den konkreten Anwendungsfall zugeschnitten werden und einfache Funktionen berechnen um mit akzeptabler Performance realisiert werden zu können. Enklaven (oft verkauft unter den Begriffen “Trusted Computing” und “Confidential Computing”) behaupten, sichere Berechnung mit nur minimalen Performance-Einbußen zu ermöglichen, im Vergleich zur Berechnung auf gewöhnlicher Hardware. Abhängig von der konkreten Umsetzung behaupten Enklaven das Programm von aller anderen Software auf dem System und sogar vor physischen Angriffen zu isolieren. Schlussendlich bieten Enklaven einen Mechanismus, um das Ergebnis einer Ausführung zu “attestieren”. Damit wird anderen ermöglicht, zu prüfen, dass die erwartete Berechnung in einer Enklave des jeweiligen Herstellers durchgeführt wurde und zu diesem Ergebnis geführt hat. Ein Nachteil dieses Ansatzes besteht darin, dass die Integrität und Vertraulichkeit der Berechnung ausschließlich von der Annahme abhängt, dass Enklaven dieses Herstellers nicht erfolgreich angegriffen werden können. Eine einzige unsichere Enklave reicht aus, um “attestations” zu fälschen und die Berechnung anzugreifen. Im Gegensatz dazu benötigt MPC nur Vertrauen in die eigene Hardware und die Sicherheit der verwendeten kryptografischen Protokolle.

Diese Dissertation zeigt, wie man die Vorteile von MPC und Enklaven verbinden kann, und damit Protokolle erhält, die bessere Performance und/oder bessere Sicherheit wie reine MPC Protokolle bieten, ohne vollständig von dem Vertrauen in die verwendeten Enklaven abzuhängen. Sie zeigt auch, dass dieses Vertrauen keine binäre Wahl ist, sondern abhängig vom Grad des Vertrauens ein Protokoll verschiedene Sicherheits-Eigenschaften haben kann, und verschiedene Performance-Verbesserungen möglich sind. Zusätzlich wird gezeigt, dass das vollständige Vertrauen in die eigene Hardware, wie es bei MPC-Protokollen oft angenommen wird, nicht nötig ist, und sichere Berechnungen trotzdem realisiert werden können, selbst wenn der eigenen Hardware nicht vollständig vertraut wird.

Im Einzelnen haben wir das zur Zeit schnellste Protokoll für Private Set Intersection konstruiert und dabei nur Enklaven mir reduzierten Vertrauensannahmen verwendet. Neben seinen vielfältigen Anwendungen ist Private Set Intersection ein interessantes Problem für diese Untersuchung,

da hierfür viele spezialisierte und hoch performante MPC Protokolle existieren, was es zu einem interessanten Vergleich macht. Unsere Konstruktion senkt die Performancekosten weiter, indem Enklaven genutzt werden, wobei hier nur recht geringes Vertrauen in deren Sicherheit angenommen wird. Für eine andere Anwendung, förderiertes Lernen von Entscheidungsbäumen, konstruieren wir ein spezifisches MPC Protokoll, das aber gewisse Menge an Information nicht geheim hält. Wir zeigen, wie dessen Sicherheit durch Enklaven verstärkt werden kann, was zu verschiedenen Sicherheitsleveln führt, abhängig davon, welche Sicherheitsannahme man bereit ist, für die Enklave zu treffen. Wenn die Enklaven komplett sicher sind, hält das verbesserte Protokoll alle Information geheim, wenn die Enklaven Speicherzugriff-Seitenkanäle haben, schützt das Protokoll eine gewisse Menge an Information nicht, und wenn die Enklaven schlechtest-mögliche Seitenkanäle haben, ist der Verlust an Geheimhaltung, wie ohne die Verwendung von Enklaven. Zusätzlich zeigt diese Dissertation, dass sogar das normalerweise angenommene Vertrauen in die eigene Hardware nicht einfach gegeben werden muss. Wir zeigen, wie sichere Berechnung mit noch weniger Vertrauensannahmen realisiert werden kann, als sie für normale MPC benötigt werden, auf Kosten von Performance. Für dieses Ergebnis nehmen wir eine beliebige Ausführung einer virtuellen Maschine und zeichnen nicht-deterministisches Verhalten auf. Ein zweites System verifiziert dann diese Berechnung (möglicherweise von einem anderen Hersteller), wodurch Sicherheit erreicht wird, wenn sich nur eines der zwei Systeme ehrlich verhält.

Contents

List of Figures

List of Tables

Part I.

Introduction

1. Introduction

1.1. Motivation

Joint computation is omnipresent in today's connected society. Computer systems are rarely isolated and share data with other systems with some expectation of the kind of processing the receiver performs with the data. This reaches from managed cloud services (end-user applications, infrastructure services) over communication services (messaging services, online shopping) to batch processing of larger data (data analysis, machine learning). The indiviual parties' roles and responsibilities are usually handled in contracts. In Europe the GDPR regulates the handling of personal data by organizations and the required contracts between parties to establish their roles in processing the data. However the guarantees that can be derived from such contracts are pretty weak. In particular, for a contract to have any consequence, a breach of contract needs to be provable in court, which can get complex if not impossible depending on the situation.

Cryptographic tools can be used to restrict the processing of transmitted data. In its simplest form, for transport encryption, it allows to prevent a transmitting entity from learning or modifying the data. However in principle, cryptographic tools can be used to restrict the computation to an arbitrary function. This primitive is called secure multi-party computation (MPC). For arbitrary functions this usually incurs a significant performance penalty. Specialized protocols realizing MPC with lower performance penalty for specific functions exist for some functions.

Enclaves claim to realize secure computation of arbitrary functions with minimal performance penatly. In order to achieve this, enclaves attest their output, together with their state, to convince a relying party that the enclave is running the desired program. After verifying this attestation, the relying party can transmit their data, trusting that the enclave will follow the given program and only perform the desired function. The relying party has no guarantee, and often not even a means to verify, that the enclave behaves as claimed by the manufacturer. This requires trust in the manufacturer of the enclave, as a successful attack on the enclave could now forge an attestation or read out the interial state of the enclave. Depending on the surrounding protocol, this can reveal the complete input data.

1.2. Structure of this Thesis

After clarifying some foundations about cryptographic primitives in section 1.3, we will look at two specific applications (Private Set Intersection chapter 2, and Decision Tree Training chapter 3) with application-specific MPC protocols with and without enclaves. In both applications the enclaves allow using some amount of hardware trust to enable more efficient MPC protocols, giving different security guarantees. Subsequently, in chapter 4, we will look at a way to realize computation without even trusting one's own system to execute its program as intended. Finally

we conclude this thesis by characterizing the relationship between hardware trust assumptions, performance, and resulting security guarantees, that these applications have shown.

1.3. Preliminaries

This section builds upon the preliminaries of the main chapters of this thesis ([42, 13, 72]).

1.3.1. Notation

PPT refers to a probabilistic algorithm with a polynomial runtime bound. For a probabilistic algorithm X, let $x \leftarrow X(a)$ denote the output of X on input a. For a set S, let $s \xleftarrow{\$} S$ denote that s is chosen uniformly at random from S. $\kappa \in \mathbb{N}$ denotes a (computational) security parameter, $\lambda \in \mathbb{N}$ a statistical one. When discussing hybrid games for a security proof using game hopping, for a hybrid H_i, let out_i denote its output.

Furthermore, $(\mathbb{G}, p, [1])$ is an additive group of prime order $p > 3$. We use the additive implicit notation for group operations with the group generator $[1]$. In implicit notation, $x \cdot [y] = [x \cdot y]$. In this notation, the DDH assumption means that $([1], [x], [y], [x \cdot y])$ and $([1], [x], [y], [z])$ are computationally indistinguishable for $x, y, z \leftarrow \mathbb{Z}_p^\times$.

A function $\mathsf{PRF}\colon \mathbb{Z}_p \times \{0,1\}^* \to \mathbb{G}$ is a pseudorandom function (PRF), if oracle access to PRF with a random key $k \leftarrow \mathbb{Z}_p$ is computationally indistinguishable to oracle access to a random function.

1.3.2. Introduction to Universal Composability

In the following, we give a brief overview of the Universal Composability (UC) security notion. The following is adapted from [19]. For detailed discussions, see [23, 22, 8].

In the UC framework, security is defined by the indistinguishability of two experiments: the ideal experiment and the real experiment. In the ideal experiment, the task at hand is carried out by dummy parties with the help of an ideal incorruptible entity—called the ideal functionality $\mathcal{F}$. In the real experiment, the parties execute a protocol π in order to solve the prescribed task themselves. A protocol π is said to be a (secure) realization of $\mathcal{F}$ if no PPT machine $\mathcal{Z}$, called the environment, can distinguish between these two experiments. In contrast to previous simulation-based notions, indistinguishability must not only hold after the protocol execution has completed, but even if the environment $\mathcal{Z}$—acting as the interactive distinguisher—takes part in the experiment, orchestrates all adversarial attacks, gives input to the parties running the challenge protocol, receives the parties' output and observes the communication during the whole protocol execution.

UC Framework Conventions. In the UC framework, each party is identified by its party identifier (PID) pid which is unique to the party and is the UC equivalent of the physical identity of this party. A party runs a protocol π is called the main party of this instance of π. Let μ denote the code of parties of π. A subsidiary and its parent use their input/subroutine output tape to communicate with each other. The set of machines taking part in the same protocol but for different parties communicate through their backdoor tapes via the adversary. An instance of a protocol is identified by its session identifier (SID) sid. All machines taking part in the same protocol instance share the same SID. A specific machine is identified by unique its extended identity $(\mu, \text{sid}||\text{pid})$.

The (Dummy) Adversary. The adversary $\mathcal{A}$ is instructed by the environment and represents the environment's interface to the execution. To this end, all messages from any party to a party that has a different main party and that are intended to be written to an incoming message tape are copied to the adversary. The adversary can process the message arbitrarily. The adversary may decide to deliver the message (by writing the message on its own outgoing message tape), postpone or completely suppress the message, inject new messages or alter messages in any way including the recipient and/or alleged sender.

the environment may also instruct the adversary to corrupt a party. In this case, the adversary takes over the position of the corrupted party, reports its internal state to the environment and from then on may arbitrarily deviate from the protocol in the name of the corrupted party (as requested by the environment). This means whenever the corrupted machine would have been activated (even due to subroutine output), the adversary gets activated with the same input.

A special case for the adversary is the so-called dummy adversary that reports all received messages to the environment and delivers all messages coming from the environment. It can be shown that the dummy adversary is complete, i.e. that if a simulator for the dummy adversary exists, then there also exists a simulator for any other adversary.

Ideal Functionalities and the Ideal Protocol. An ideal functionality $\mathcal{F}$ is a special type of entity whose instances bear a SID but no PID. Hence, it is an exception to the aforementioned identification scheme. Input to and subroutine output from $\mathcal{F}$ is performed through dummy parties. Dummy parties merely forward their input to the input tape of $\mathcal{F}$ and subroutine output from $\mathcal{F}$ to their own outgoing message tape. They share the same SID as $\mathcal{F}$, but additionally have individual party identifiers (PIDs) as if they were the actual main parties of a (real) protocol. The ideal functionality $\mathcal{F}$ is simultaneously a subroutine for each dummy party and conducts the prescribed task. IDEAL($\mathcal{F}$) is called the (ideal) protocol for $\mathcal{F}$ and denotes the set of $\mathcal{F}$ together with its dummy parties.

The UC Experiment. Let π be a protocol, $\mathcal{Z}$ an environment and $\mathcal{A}$ an adversary. Let ξ be an (efficiently computable) identity bound. The UC experiment, denoted by $\text{Exec}_{\pi,\mathcal{A},\mathcal{Z}}(n, a)$, initially activates the environment $\mathcal{Z}$ with security parameter 1^n and input $a \in \{0,1\}^*$. The first machine that is invoked by $\mathcal{Z}$ is the adversary $\mathcal{A}$. All other parties invoked by $\mathcal{Z}$ are set to be main parties of the challenge protocol π. $\mathcal{Z}$ freely chooses their input, their PIDs and the SID of the challenge protocol. When giving input, the environment is restricted to using extended identities such that ξ is satisfied at every point of the execution. The experiment is executed as outlined above.

Definition of Security. Let π, ϕ be protocols. π emulates ϕ in the UC framework, denoted by $\pi \geq \phi$with respect to ξ-bounded environments, if for every PPT adversary $\mathcal{A}$ there is a PPT adversary $\mathcal{S}$ such that for every ξ-identity-bounded PPT environment $\mathcal{Z}$, there is a negligible function negl such that for all $n \in \mathbb{N}, a \in \{0,1\}^*$ it holds that

$$|\Pr[\mathrm{Exec}(\pi, \mathcal{A}, \mathcal{Z})(n, a) = 1] - \Pr[\mathrm{Exec}(\phi, \mathcal{S}, \mathcal{Z})(n, a) = 1]| \leq \mathsf{negl}(n)$$

where $\mathrm{Exec}(\pi, \mathcal{A}, \mathcal{Z})(n, a)$ denotes the random variable for the environment $\mathcal{Z}$'s output in the UC execution experiment with protocol π and adversary $\mathcal{A}$ on input a and security parameter n.

The simulator $\mathcal{S}$ mimics the adversarial behavior to the environment as if this were the real experiment with real parties carrying out the real protocol with real π-messages. Moreover, $\mathcal{S}$ must come up with a convincing internal state upon corrupted parties, consistent with the simulated protocol execution up to this point (dummy parties do not have an internal state).

Protocol Composition. UC security is closed under protocol composition: Let π, ϕ, be subroutine-respecting protocols[1] and let ρ be an arbitrary protocol.

$$\pi \geq \phi \implies \rho^{\pi} \geq \rho^{\phi}$$

Global Ideal Functionalities via UC with Global Subroutines Initially, UC security was not designed to deal with global ideal functionalities, i.e. ideal functionalities that are used by multiple protocols. Generalized UC (GUC) security [24] extended UC security to capture security in the rpesence of global subroutines such as global ideal functionalities. Unfortunately, GUC security suffers from several technical problems [8].

As a consequence, UC security has been extended to allow the modeling of global ideal functionalities within UC security [8, 22]. In more detail, the Universal Composition with Global Subroutines (UCGS) formalism captures the execution joint of a protocol π (as in UC) and a global ideal functionality $\mathcal{G}$ within a single so-called management protocol $\mathsf{M}[\pi, \mathsf{SIM}^{\mathsf{ORE}}_{\mathcal{A},\mathcal{S},(}\mathcal{G})]$. As $\mathsf{M}[\cdot]$ is a UC protocol, this formalism recovers all properties of UC security.

We restate the definition of UC emulation with global subroutines.

Definition 1 (UC emulation with global subroutines [8]) Let π, ϕ and γ be protocols. We say that pi ξ-UC-emulates ϕ in the presence of γ if protocol $\mathsf{M}[\pi, \gamma]$ ξ-UC-emulates protocol $\mathsf{M}[\phi, \gamma]$.

1.3.3. Building Blocks and Security Notions

Deterministic Secret-Key Encryption. In our protocol, we use deterministic stateless secret-key encryption schemes (SKE). It is well known that such encryption schemes cannot fulfill the established notion of indistinguishability under chosen plaintext attack (IND-CPA) security: Upon sending challenges (m_0, m_1) and receiving a ciphertext c, the adversary can determine if c is associated with m_0 or m_1 by simply querying its encryption oracle on m_0 or m_1 and comparing the result with c.

[1] Very informally, a protocol π is subroutine-respecting if a machine μ of π does not communicate with a machine μ' that is not part of π or of a sub-protocol of π.

By restricting such queries, IND-CPA security can be meaningfully relaxed for the deterministic and stateless setting. Instead of using the "find-then-guess" variant of IND-CPA as in the example above, we propose a notion based on the "left-or-right" (LoR) variant of IND-CPA security [11] (also in [17]), which is often more convenient in reductions and equivalent to the find-then-guess variant. In LoR-CPA, the adversary is not given a single challenge ciphertext c, but interacts with a LoR oracle that can be queried multiple times on ciphertext pairs (m_0^i, m_1^i). Depending on the choice bit b, the adversary always receives encryptions of m_b^i.

Definition 2 (det-LoR-CPA Security for SKE) Let $\mathsf{SKE}_{\mathsf{detCPA}} = (\mathit{Gen}, \mathit{Enc}, \mathit{Dec})$ be a deterministic SKE scheme and let $\mathrm{Exp}^{\text{det-LoR-CPA}}_{\mathcal{A},\mathsf{SKE}_{\mathsf{detCPA}}}(\kappa, z)$ denote the output of the following experiment:

Experiment $\mathrm{Exp}^{\text{det-LoR-CPA}}_{\mathcal{A},\mathsf{SKE}_{\mathsf{detCPA}}}(\kappa, z)$:

1. Define $\mathcal{U}_{\mathit{Enc}}(k, b, m_0, m_1)$ as $\mathit{Enc}(k, m_b)$.
2. $k \leftarrow \mathit{Gen}(1^\kappa)$
3. $b \xleftarrow{\$} \{0,1\}$
4. $b' \leftarrow \mathcal{A}(1^\kappa, z)^{\mathcal{U}_{\mathit{Enc}}(k,b,\cdot,\cdot)}$
5. Return 1 if $b = b'$ and 0 otherwise.

We say that $\mathsf{SKE}_{\mathsf{detCPA}}$ is deterministic left-or-right-secure under chosen-plaintext attack (det-LoR-CPA-secure) if for every legal PPT adversary $\mathcal{A}$, there exists a negligible function negl such that for all $\kappa \in \mathbb{N}$ and all $z \in \{0,1\}^*$, it holds that $\mathrm{Adv}^{\text{det-LoR-CPA}}_{\mathcal{A},\mathsf{SKE}_{\mathsf{detCPA}}} := |\Pr[\mathrm{Exp}^{\text{det-LoR-CPA}}_{\mathcal{A},\mathsf{SKE}_{\mathsf{detCPA}}}(\kappa, z) = 1] - \frac{1}{2}| \leq \mathsf{negl}(\kappa)$.

Let Q be the set of queries sent by $\mathcal{A}$ to $\mathcal{U}_{\mathit{Enc}}$. An adversary $\mathcal{A}$ is legal if it holds that

- $|m_0^i| = |m_1^i|$ for all $(m_0^i, m_1^i) \in Q$ and
- $m_0^i = m_0^j \iff m_1^i = m_1^j$ for all i, j in $\{1, \ldots, |Q|\}$.

Looking ahead to our construction, we only reduce to the security of ciphertexts created by honest parties. Thus, the adversary has no (implicit) access to an encryption oracle. As such, the restrictions in definition 2 for the adversary are natural in the considered setting and do not require additional assumptions about the adversary.

Key Exchange. Key exchange allows parties to exchange a secret key in the presence of an adversary. In the following, we re-state the definition of universally composable key exchange from [29]. This definition can be satisfied by a variant of the well-known Diffie-Hellman key exchange [39] if authenticated communication is available [29].

Definition 3 (Ideal Functionality $\mathcal{F}_{\mathrm{KE}}$ (adapted from [29])) $\mathcal{F}_{\mathrm{KE}}$ proceeds as follows, running on security parameter κ, with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{S}$.

Upon receiving an input $(\texttt{Establish-session}, \mathrm{sid}, P_i, P_j, \mathrm{role})$ from some party P_i, record the tuple $(\mathrm{sid}, P_i, P_j, \mathrm{role})$ and send this tuple to the adversary. In addition, if there already is a recorded tuple $(\mathrm{sid}, P_j, P_i, \mathrm{role}')$ (either with $\mathrm{role}' \neq \mathrm{role}$ or $\mathrm{role} = \mathrm{role}'$) then proceed as follows:

1. If both P_i and P_j are uncorrupted then choose $k \xleftarrow{\$} \{0,1\}^\kappa$ and make a private delayed output $(\texttt{key}, \text{sid}, k)$ to P_i and P_j.
2. If either P_i or P_j is corrupted, then send a message $(\texttt{Choose-value}, \text{sid}, P_i, P_j)$ to the adversary; receive a value k from the adversary, make outputs $(\texttt{key}, \text{sid}, k)$ to P_i and P_j and halt.

Universal Hash Functions. Universal hash functions (UHFs) [30] allow the compression of elements with a bounded collision probability. In contrast to cryptographic hash functions, collision resistance is only guaranteed if the UHF is chosen after the elements to be hashed. However, UHFs exist unconditionally, i.e. no complexity assumptions are required.

Definition 4 (Universal Hash Function) We say that a family of hash functions $H_\lambda = \{h : X_\lambda \to Y_\lambda\}$ is universal if for all $x_1, x_2 \in X_\lambda$ such that $x_1 \neq x_2$, it holds that $\Pr[h(x_1) = h(x_2) \mid h \leftarrow H_\lambda] \leq 1/|Y_\lambda|$, where the probability is over the choice of h.

We usually consider Y to depend on a statistical security parameter λ instead of a computational security parameter κ.

1.3.4. Key-homomorphic PRF

A PRF is key-homomorphic, if for all messages $m \in \{0,1\}^*$ and keys $k, k' \in \mathbb{Z}_p$

$$\mathsf{PRF}(k, m) + \mathsf{PRF}(k', m) = \mathsf{PRF}(k + k', m)$$

and therefore also

$$a \cdot \mathsf{PRF}(k, m) = \mathsf{PRF}(a \cdot k, m)$$

for all $a \in \mathbb{Z}_p$.

Naor, Pinkas, and Reingold [83] have constructed a key-homomorphic PRF under the DDH assumption in the random oracle model for $\mathsf{RO}\colon \{0,1\}^* \to \mathbb{G} \setminus \{[0]\}$:

$$\mathsf{PRF}(k, m) := k \cdot \mathsf{RO}(m) \tag{1.1}$$

1.3.5. Order-Revealing Encryption

We follow the definition in [35]. An ORE scheme is defined as a 3-tuple of PPT algorithms $\mathsf{ORE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Cmp})$ over message space $\mathcal{M}$, key space $\mathcal{K}$ and ciphertext space $\mathcal{C}$, where

- $\mathsf{Gen}(1^\kappa)$ returns a secret key $k \in \mathcal{K}$
- $\mathsf{Enc}(k, m)$ takes a key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$ as input and returns a ciphertext $c \in \mathcal{C}$
- $\mathsf{Cmp}(c_0, c_1)$ is deterministic, takes two ciphertexts $ct_0, ct_1 \in \mathcal{C}$ and returns a bit b or $\perp$.

$\mathsf{REAL}^{\mathsf{ORE}}_{\mathcal{A}}(\kappa)$:
1: $k \leftarrow \mathsf{Gen}(1^\kappa)$
2: $m \leftarrow \mathcal{A}(1^\kappa)$
3:
4: $ct_1 \leftarrow \mathsf{Enc}(k, m_1)$
5: for $i = 2, \ldots, q$ do
6: $\quad m \leftarrow \mathcal{A}(ct_1, \ldots, ct_{i-1})$
7:
8: $\quad ct_i = \mathsf{Enc}(k, m)$
9: end for
10: output $(ct_1, \ldots, ct_q)$ and the state of $\mathcal{A}$

$\mathsf{SIM}^{\mathsf{ORE}}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\kappa)$:
1: $M = \emptyset$
2: $m \leftarrow \mathcal{A}(1^\kappa)$
3: M.append(m)
4: $ct_1 \leftarrow \mathcal{S}^{\mathsf{ORE}}(\mathcal{L}(M))$
5: for $i = 2, \ldots, q$ do
6: $\quad m \leftarrow \mathcal{A}(ct_1, \ldots, ct_{i-1})$
7: $\quad M$.append(m)
8: $\quad ct_i \leftarrow \mathcal{S}^{\mathsf{ORE}}(\mathcal{L}(M))$
9: end for
10: output $(ct_1, \ldots, ct_q)$ and the state of $\mathcal{A}$

Figure 1.1.: Definition of experiments $\mathsf{REAL}^{\mathsf{ORE}}_{\mathcal{A}}$ and $\mathsf{SIM}^{\mathsf{ORE}}_{\mathcal{A},\mathcal{S},\mathcal{L}}$ for ORE scheme ORE and stateful TMs $\mathcal{A}$ and $\mathcal{S}^{\mathsf{ORE}}$, cf. [35].

We require correctness for the scheme:

$$\forall m_0, m_1 \in \mathcal{M}, k \leftarrow \mathsf{Gen}(1^\kappa): \\ \Pr[\mathsf{Cmp}(\mathsf{Enc}(k, m_0), \mathsf{Enc}(k, m_1)) = 1 \Leftrightarrow m_0 < m_1] \geq 1 - \mathsf{negl}(\kappa).$$

In addition, ORE schemes have an implicit $\mathsf{Dec}(k, c)$ algorithm, which takes a secret key $k \in \mathcal{K}$ and a ciphertext $c \in C$ and outputs a message $m \in \mathcal{M}$. Dec is implicitly defined using Enc and Cmp to perform a binary search on the message space and return the result or $\perp$ if the ciphertext is invalid under key k.

Security of ORE Schemes For the security definition of an ORE schemes w.r.t. some leakage function $\mathcal{L}(m_1, \ldots, m_t)$, we use the security notion defined in [35]. Let $\mathsf{ORE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Cmp})$ be an ORE scheme. For some $q = \mathsf{poly}(\kappa)$, let $\mathcal{A}$ be a stateful adversary and let $\mathcal{S}^{\mathsf{ORE}}$ be a stateful simulator. Then, the experiments $\mathsf{REAL}^{\mathsf{ORE}}_{\mathcal{A}}$ and $\mathsf{SIM}^{\mathsf{ORE}}_{\mathcal{A},\mathcal{S},\mathcal{L}}$ are defined as in fig. 1.1. We say an ORE scheme ORE is secure w.r.t. the leakage function $\mathcal{L}$, if there exists a PPT simulator $\mathcal{S}^{\mathsf{ORE}}$, such that for all PPT adversaries $\mathcal{A}$, the games $\mathsf{REAL}^{\mathsf{ORE}}_{\mathcal{A}}$ and $\mathsf{SIM}^{\mathsf{ORE}}_{\mathcal{A},\mathcal{S},\mathcal{L}}$ are computationally indistinguishable.

Best-Possible Leakage for ORE schemes Naturally, one is interested in an ORE scheme with the least amount of leakage. Given t ORE ciphertexts $c_1, \ldots, c_t \in C$ of messages $m_1, \ldots, m_t \in \mathcal{M}$ under the same key, then for each pair (c_i, c_j), one can inevitably learn whether $m_i \leq m_j$ by using the comparison algorithm. We call a leakage function $\mathcal{L}_{\text{ideal}}$ best-possible or ideal, if one learns nothing else, i.e., if the leakage is given by

$$\mathcal{L}_{\text{ideal}}(m_1, \ldots, m_t) = \{(i, j) \mid m_i \leq m_j\}.$$

Note that unfortunately there is no known ORE scheme for superpolynomial message space with this ideal leakage, except for one that uses very strong assumptions, such as multilinear maps [16], rendering this scheme unsuitable in practice.

Assumptions on the Leakage Functions In the remainder of this work, we make a few assumptions about the leakage function.

Assumption 1 $\forall m : \mathcal{L}(m) = \emptyset$.

While one could think of ORE schemes, for which this assumption does not hold, the assumption holds for established schemes like the ones in [35, 70], as well as our ORE scheme.

Assumption 2 For t messages $m_1, \ldots, m_t$ and index $0 \leq i \leq t$, it holds that $L^{\leq i} \coloneqq \mathcal{L}(m_1, \ldots, m_i)$ can be efficiently computed from $L = \mathcal{L}(m_1, \ldots, m_t)$.

While this assumption does not have to hold for all ORE leakage functions, it does hold for leakage functions that tightly capture the leakage of the respective ORE scheme. Again, this assumption holds for the ORE schemes in [35, 70], as well as for our ORE scheme.

1.3.6. Decision Tree Training

We consider a domain with data points x with X continuous attributes x_j and a label $\ell(x)$ from a small discrete set of labels. In this context, a decision tree is a binary tree, where each leaf node contains a label and each inner node contains a tuple (j, t), where j is an index of an attribute and t is a value of the j-th attribute. When performing a classification on a data point x at an inner node (j, t), we recurse to the left child node, if $x_j \leq t$, and to the right child node, otherwise. This is repeated until reaching a leaf node, where the label of the node is returned.

Decision tree training is the task of building a decision tree given a set of labeled training data. Established decision tree frameworks like [20, 1] use variations of the recursive C4.5 algorithm by Quinlan [91]. An adaption of this algorithm can be seen in algorithm 1. One source of variation in this algorithm is the heuristic H used in algorithm 1. A common example used here is the Information Gain heuristic. Following the definition in [65], it is defined as

$$H'(S) = -\sum_{l \in \mathbb{L}} \frac{|\{x \in S \colon \ell(x) = l\}| \log(|\{x \in S \colon \ell(x) = l\}|)}{|S|},$$

$$H(S, L, R) = H'(S) - \frac{|L|}{|S|} H'(L) - \frac{|R|}{|S|} H'(R),$$

where $\mathbb{L}$ is the set of labels. This heuristic describes the information-theoretic gain when separating the set S into sets L and R. Another common heuristic is the GINI Index, which performs nearly identically as the information gain heuristic [101]. The other common variation is the threshold value used in algorithm 1, where most frameworks use an intermediate value between the threshold and the next larger occurring attribute value. While in a classical setting both the labels as well as the attribute values, are numeric values, we note that the training algorithm itself only needs to evaluate the order of attribute values. This is required in algorithm 1. While one could think of a heuristic H which requires additional operations on the attributes, established heuristics like Information Gain do not consider the attribute values, but only take the labels into consideration.

It is possible to use different training algorithms and variations of trees, such as XBoost [34], which trains gradient-boosted trees. In this approach, the training process requires performing

Algorithm 1 Decision tree training with a heuristic H.

```
1: function TrainDecisionTree(data)
2:     assert ∀x, y ∈ data : ∃j : x_j ≠ y_j
3:     if ∀x ∈ data: ℓ(x) = ℓ(data_0) then
4:         return LeafNode(ℓ(data_0))
5:     end if
6:     j* := thresh* := ⊥, h* := −∞
7:     for 1 ≤ j ≤ X do
8:         L := ∅
9:         for thresh ∈ {x_j | x ∈ data} in ascending order do
10:            L := L ∪ {x ∈ data | x_j = thresh}
11:            h := H(data, L, data \ L)
12:            if h > h* then
13:                j* := j, thresh* := thresh, h* := h
14:            end if
15:        end for
16:    end for
17:    L := {x ∈ data | x_j* ≤ thresh*}, R := data \ L
18:    return InnerNode(j*, thresh*, TrainDecisionTree(L), TrainDecisionTree(R))
19: end function
```

arithmetic operations on the attributes. Another variation is the use of decision forests consisting of multiple decision trees. A classification of a data point in such a decision forest is done by classifying it with each tree and performing a majority vote on the resulting labels. Bentéjac, Csörg, and Martnez-Muñoz [12] have empirically shown that in a real-world scenario, gradient boosted trees, although requiring arithmetic operations on the training data, do not perform significantly better than decision forest that can be trained with algorithm 1 only comparing elements and performing equality checks on the labels.

Part II.

Applications

2. Private Set Intersection

This chapter is based on joint work with Jeremias Mechler and Prof. Jörn Müller-Quade published at ASIACRYPT 2023[42]. My main contribution in this work was the core idea, that assuming remote attestations are secure, is not far off from assuming elementary cryptographic building blocks are also secure. This is also inspired by the way Intel SGX implements remote attestations. The implementation of the proposed protocol and evaluating its performance was also primarily done by me.

2.1. Introduction

Private set intersection (PSI) allows mutually distrusting parties $P_1, \ldots, P_n$ with input sets $S_1, \ldots, S_n$ to compute the intersection $S = S_1 \cap \cdots \cap S_n$ such that a dishonest party does not learn anything about another party's set that it cannot compute from its own input and the intersection S.

PSI can be used for a number of problems, making it an important privacy-enhancing technology:

- malware [99, 100] or spam [48] detection, where a server holds a list of signatures against which a client wants to check an email or an executable,
- recognition of child pornography, for example on mobile devices or on cloud storage, either by direct comparison or through perceptual hashing [66],
- contact discovery for messenger services [78],
- COVID contact tracing [44] or
- learning if secret agents have been arrested [7].

In many of these cases, it is not only suffices, but highly desirable for privacy reasons if only one party learns the intersection result. This natural variant, called one-sided PSI [93], has been widely considered in the literature (e.g. [7, 55, 93, 90, 47, 92]).

Security Notions for PSI. For a long time, the security of PSI protocols has mainly been considered for the case of passive corruptions in a stand-alone setting, i.e. for semi-honest adversaries that adhere to the protocol description and without the presence of other protocol executions. While this setting allows (relatively) efficient protocols, the offered security is often insufficient: In real life, one or more of the protocol parties may behave maliciously with the intent of gaining information about another party's secrets. Thus, it is very important that PSI protocols are secure even against malicious adversaries. Most recent works like [90, 92] have both a semi-honest and a malicious variant with the malicious variant being a bit slower as it has to perform additional consistency checks.

Typically, protocols are seldom executed in isolation. When PSI is used for contact discovery by a messenger on a smartphone, other apps are executed in the background and may also execute cryptographic protocols, e.g. the establishment of TLS sessions. These other cryptographic protocols should not affect the security of the PSI, and vice versa.

In order for security to hold in such a setting, an appropriate security notion that is closed under general concurrent composition, i.e. holds in the presence of arbitrary other protocols that are executed concurrently, is necessary. This is fulfilled by Universally Composable (UC) security [23].

Composable Security. UC security is based on the simulation paradigm where the "real-world" execution of a protocol π is compared to the "ideal-world" execution of an ideal functionality $\mathcal{F}$, which acts as a trusted third party performing the desired task by definition. For every real-world adversary $\mathcal{A}$ in the execution with π, the existence of a corresponding ideal-world adversary $\mathcal{S}$ interacting with $\mathcal{F}$, called the simulator, must be proven. Both executions must be indistinguishable for an interactive distinguisher $\mathcal{Z}$, called the environment. The environment (adaptively) chooses the parties' inputs and may communicate with the adversary freely throughout the execution. If $\mathcal{Z}$ cannot distinguish between the real and the ideal execution, then all properties of the ideal execution, which is secure by definition, carry over to the real execution. In contrast to stand-alone security notions, this strong security notion is harder to achieve.

In order to achieve UC security, several prerequisites have to be met:

1. A trusted setup such as a common reference string, a public key infrastructure or a random oracle is necessary [25, 24].
2. The simulator has to be able to extract the inputs of a corrupted party.

For a large class of setups such as a common reference string or a public-key infrastructure, a protocol's communication complexity is lower-bounded by the size of a party's input (implicit in [81]). Unlike in protocols with stand-alone security, we thus cannot hope to compress the set elements by simply (locally) applying a cryptographic hash function, as this would prevent extraction.

Use of Random Oracles in Previous Protocols. In order to circumvent this extractability problem and still allow compression, many protocols for composable PSI (e.g. [93, 90, 47, 92]) resort to the use of random oracles, which can be thought of as idealized hash functions. As they provide input awareness, extraction and thus composable security becomes possible.

However, all security may be lost when the random oracle is instantiated by e.g. a cryptographic hash function [27, 26]. Thus, there usually is a large gap between the security proof and the actual security provided by the modified protocol used in the implementation. Moreover, security proofs may use properties such as code-correlation robustness (e.g. in [90, 47]) provided by the random oracle, which also may not be fulfilled by its instantiation.

Additional and more subtle problems may arise during protocol composition, as several protocol instances may implicitly share state via a common hash function[1] that is used to instantiate the

[1] While this can be prevented in principle by using appropriate building blocks (e.g. by obtaining an independent key for the hash function using a coin-toss build from a non-malleable commitment scheme), we are not aware that this is done in practice.

random oracle, possibly allowing malleability attacks. This also holds for global random oracles [28, 21], unless special care to obtain the session ID or hash key is taken. Thus, the gap between the protocol with security proof and the implemented protocol may be even larger.

Hardware Assumptions as Alternatives. In order to close the gap between the protocol under analysis and the protocol to be implemented while still achieving high performance, in particular with network communication complexity that is independent of the elements' size, we investigate how hardware assumptions can be leveraged to achieve composable PSI.

Trusted execution environments (TEEs), which have been available for several years on mainstream CPUs, promise the secure execution of user-supplied programs in isolation from the host system at near native speed. In particular, some modern TEEs have a feature called remote attestation, which enables a third party to obtain evidence about the code running inside a TEE. This is sufficient to establish a cryptographic secret with the TEE. Using this secret, private inputs can be passed into the TEE for further processing. Due to the isolation properties of the TEE, secrets are assumed to be protected even if the computer hosting the TEE is compromised.

With such strong assumptions, one-sided PSI for two parties seems trivial: Both parties perform remote attestation with the TEE, thereby verifying that it runs the expected code and establishing a secure channel. Via this secure channel, they send their private inputs. After the TEE has received the inputs and computed the intersection, one party obtains the output. However, it can be shown that any protocol with only one TEE cannot satisfy UC security [88, 14] for many tasks, including PSI, highlighting the technical challenges in achieving such a strong security notion.

Reducing Trust. Leaving this fact aside, the simple PSI protocol suffers from several problems. First of all, the run-time of the program executed inside the TEE may depend on a party's secret input: If one input is {0}, the computation might take one second. If it is {1}, it might take one minute. This inherent timing side-channel, which can be easily observed from the outside by waiting for the result, is not mitigated by TEEs.

Even if the run-time of the program in question is not subject to such gross timing side-channels, the TEE usually needs to be trusted completely, which may not be warranted:

- TEEs such as Intel SGX may not protect against even more subtle side-channels resulting e.g. from memory access patterns by design [37], possibly requiring expensive techniques such as oblivious RAM (ORAM) [49, 4] to mitigate them.
- They repeatedly suffered from a number of vulnerabilities exposing additional side-channels, allowing the party hosting the TEE to learn all its secrets and even impersonate a TEE [85, 104, 82, 96].

Even when ignoring vulnerabilities, side-channels that exist by design could render the above protocol completely insecure, as the party hosting the TEE could be able to learn all the inputs of the other party. Thus, measures against side-channels have to be included into the protocol design.

Transparent enclaves, introduced by Tramèr et al. [102], (also in [88]) consider this setting. Transparent enclaves are able to securely perform remote attestation, but otherwise have no secrets whatsoever with respect to their respective host, thereby capturing all possible

side-channels. Interestingly, such transparent enclaves still allow for the efficient realization of cryptographic building blocks like commitment schemes or zero-knowledge proof systems [102, 88]. This is because the transparent TEE only sees secrets belonging to its owner (and attestation secrets also remain secure, preventing a corrupted party to impersonate a TEE). For the same reason, transparent enclaves can be used as efficient passive-security-to-active-security compilers.

In line with the model of [102, 88], we consider an intermediate setting between fully trusted and fully transparent TEEs. In particular, we assume that very simple cryptographic building blocks like key exchange or secret-key encryption schemes are also implemented side-channel-free, just like the signature scheme in [88] used for remote attestation. Otherwise, the enclaves are again completely transparent. We believe that such an intermediate model is well-motivated as

- it can be plausibly realized (see section 2.2.4 for a discussion) in practice,
- provides a natural framework for the design and analysis of protocols using cryptography (or possibly other operations on sensitive data) in a setting where side-channels may be present and
- if even the proposed simple operations cannot be performed securely, it may be plausible to assume that remote attestation is also impossible.

We call such a TEE almost-transparent.

Additionally, we consider the setting of "passively corrupted" or "semi-honest" TEEs where all TEEs leak all secrets (to a third party that does not participate in the protocol execution as a party), e.g. to the manufacturer or some government agency. In the case of Intel SGX, such fears are particularly plausible due to the use of an Intel attestation service [61] or the out-of-band communication supported by many chipsets [67]. A similar model has been proposed before [76].

Interestingly, even such a weak assumption enables to construct a protocol for one-sided PSI that is practically efficient and features a low asymptotic communication complexity.

In order to formally capture the proposed variants, we adapt the global ideal functionality for TEEs of Pass, Shi, and Tramèr [88]. Given the fact that e.g. Intel SGX instances share state via common attestation keys, it is crucial that this shared state is also captured in the model. Here, this is achieved by using a global functionality that can be used by multiple protocols, faithfully capturing subtleties that may arise from such shared state. Using our adapted functionalities, we can prove the security of our protocol in the UC framework [23, 24], using the "Universal Composability with Global Subroutines" (UCGS) [8] formalism to capture global ideal functionalities within UC.

In contrast to protocols using random oracles, we believe that the structural gap between the protocol under analysis and the protocol to be implemented is much smaller in our case as our model faithfully captures the expected security guarantees of TEEs without fully trusting them.

Looking ahead, we will demonstrate the usefulness of our model by proposing protocols for additional interesting tasks like oblivious transfer or computing the Hamming distance.

In the following, we give an informal description of our construction for PSI.

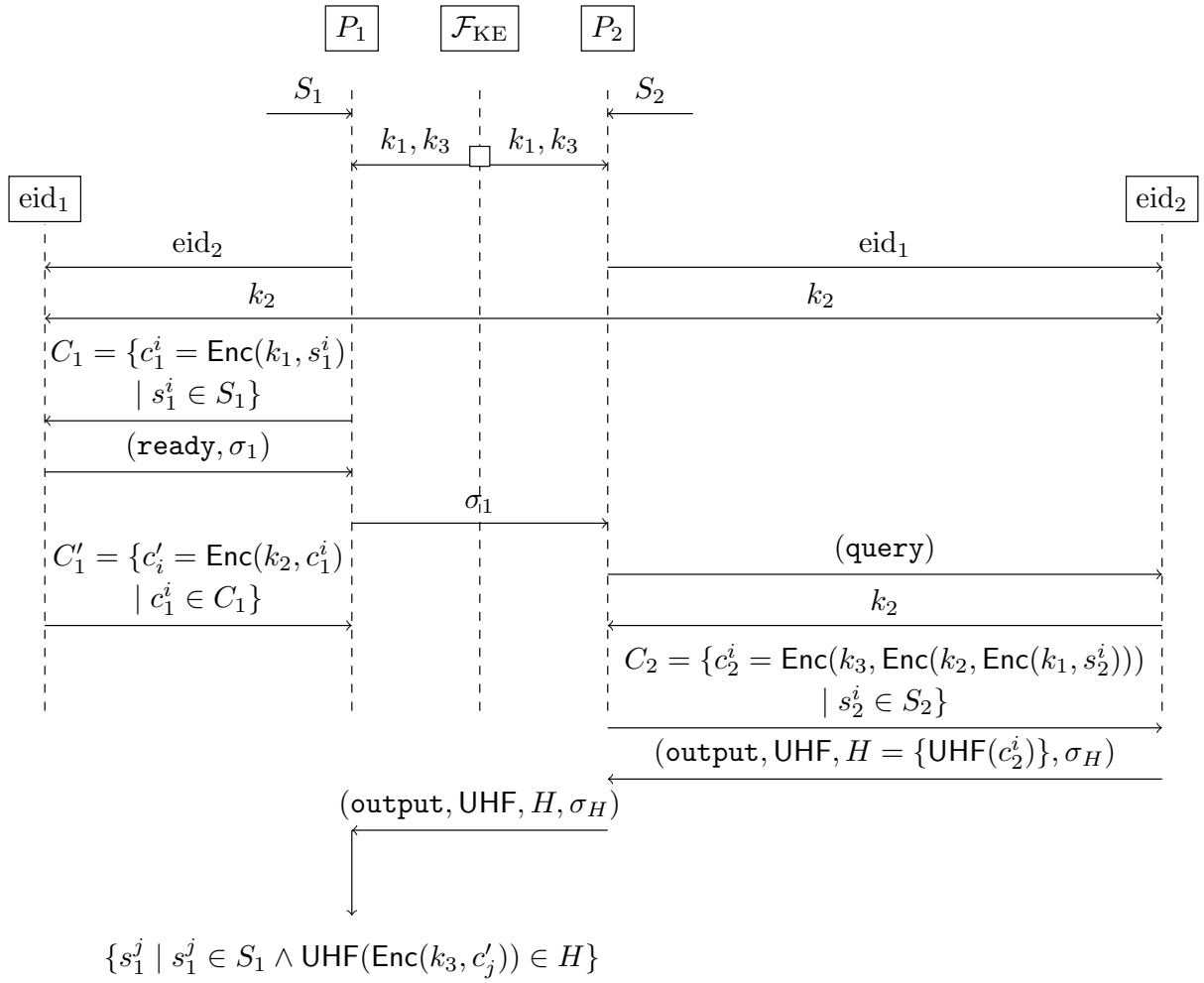

Figure 2.1.: Overview of the messages exchanged in our PSI protocol (simplified).

2.1.1. The PSI Protocol in a Nutshell

In our protocol for two-party one-sided PSI, we assume that both protocol parties P_1 and P_2 each have access to a TEE. Intuitively, we use the TEEs to

1. implement a "query-once" oracle for the party P_1 that receives the output and to
2. enforce honest behavior of the party P_2 that does not receive the intersection, but can compute arbitrary hashes (without having any hashes of P_1 to compare against).

Using their TEEs together with local computations, P_1 and P_2 both create hashes of their encrypted inputs (in order to improve communication efficiency), with P_2 eventually sending its hashes to P_1, which can then compute the intersection.

The interaction with the TEEs is done in a way such that the TEEs neither learn the parties' inputs nor the intersection's size.

Protocol Description. An overview of the messages exchanged is given in fig. 2.1. Initially, P_1 and P_2 exchange two keys k_1 and k_3 through the ideal functionality for key exchange $\mathcal{F}_{\text{KE}}$. They

use k_1 to deterministically encrypt their sets S_1 resp. S_2. P_1 then sends the resulting set of ciphertexts C_1 to its TEE, which encrypts these ciphertexts deterministically with another key k_2 that it has exchanged with the TEE of P_2, resulting in C_1'. This key k_2 can be obtained by P_2 from its TEE, but not by P_1. P_2 encrypts its ciphertexts of S_2 under k_1 iteratively also under k_2 and k_3, resulting in the set C_2. The ciphertexts in C_2, each for one element in S_2, are sorted lexicographically (to later hide the "position" of elements in the intersection) and then sent to P_2's TEE (via P_2). P_2's TEE samples a universal hash function UHF, evaluates UHF on the ciphertexts, resulting in the set H and returns UHF and H to P_2, which forwards the (signed) hashes to P_1. After having obtained the hashes H, P_1 queries its TEE to obtain C_1'. Using k_3, P_1 also encrypts the ciphertexts in C_1' with k_3, applies UHF and compares its own hashes with the hashes obtained from P_2 in H. Each element in S_1 that is associated with hashes of both parties is added to the intersection result S. Finally, P_1 sorts and outputs S. In our model, all messages coming from a TEE are digitally signed with a trusted master key and include a TEE's code and ID. Thus, such signatures can be easily recognized and checked for authenticity. Also, a party can verify that a TEE is running the expected code, which is part of the protocol description. For the formal protocol definition, see section 2.3.

Security. For an overview about which entity knows which keys, see table 2.1.

	Known to			
Key	P_1	P_2	TEE of P_1	TEE of P_2
k_1	✓	✓	-	-
k_2	-	✓	✓	✓
k_3	✓	✓	-	-

Table 2.1.: Overview of the used keys in our PSI protocol.

The intuition behind the protocol's security is as follows: If P_1 is corrupted, P_1 does not know the key k_2 and can create ciphertexts under k_2 only via its TEE. This is possible only once, as specified by the program running on the TEE. After having received hashes from P_2, P_1 is thus unable to create appropriate ciphertexts for elements not in S_1 that would, for example, allow it to perform a brute-force search to learn inputs of S_2 not in the intersection. At the same time, it cannot decrypt ciphertexts of elements of P_2 (even when ignoring the information lost due to the application of the hash function). As the hashes result from lexicographically sorted ciphertexts, a malicious P_1 with partial knowledge of P_2's input cannot learn anything about P_2's inputs not in the intersection from the order of the transmitted hashes. (If we omitted the sorting, a party P_1 with input $\{2\}$ could distinguish between inputs $\{1, 2\}$ and $\{2, 3\}$ of P_2 by observing if the first or the second hash matches.) With respect to simulation-based security, we note that the protocol does not continue unless P_1 sends its ciphertexts C_1 to the TEE. The simulator can observe this and decrypt the ciphertexts with k_1.

Conversely, if P_2 is corrupted, privacy trivially follows from the fact that P_2 does not see input-dependent messages from P_1. Extractability of a corrupted P_2's input follows from a similar argument like in the case of a corrupted P_1, together with the fact that the hash function is evaluated by the TEE, preventing P_2 from cheating after the extraction.

If P_1 and P_2 are honest, but both TEEs are passively corrupted in the sense that they leak all information (including randomness of local cryptographic operations) to a third party, we observe the following: All private inputs of P_1 to its TEE are encrypted with k_1, which the TEE does not know. The private inputs of P_2 are iteratively encrypted using keys k_1, k_2 and k_3, where k_3 is unknown to the TEEs. Thus, jointly corrupted TEEs may learn the size of each party's input, but nothing else about them. Also, the intersection's size remains hidden.

In any case, the program running on the TEEs is deterministic and does not operate on secrets, with the exception of simple cryptographic assumptions, which we assume to be secure, and, in particular, free of side channels. As a consequence, side channels (which we assume do not affect these cryptographic operations) do not affect the security of our protocol.

For the formal security proof, see section 2.3.2.

Implementation and Evaluation. We implemented our protocol with Intel SGX enclaves using a single AES operation on 128-bit blocks as deterministic encryption. For the universal hash function, we selected xxhash with a randomly chosen seed. We used the SGX remote attestation functionality to ensure that the expected enclave code is running. To verify the remote attestation, the Intel Attestation Service needs to be contacted and returns a signed confirmation of the validity of the attestation data.

We evaluated the runtime of the protocol for different input set sizes. With respect to the runtime, we distinguish between a constant input-independent setup phase and an input-dependent online phase, whose runtime is linear in the number of input elements. Computing the intersection of two sets with 2^{24} 128-bit elements takes 7.3 seconds, of which the setup phase, including communication with the Intel Attestation Service, takes 2.0 seconds.

2.1.2. Our Contribution

We present a new approach for composable private set intersection, namely the use of trusted execution environments (TEEs). This approach comes with a number of benefits, namely

- very simple protocols, facilitating an easy analysis and implementation,
- high practical performance and asymptotic complexity, with a single-threaded performance that is better than the best known protocol for (composable and maliciously secure) PSI and
- meaningful security guarantees, as the assumptions for the TEEs are comparatively weak. Moreover, there is no large gap between the protocol under analysis and the protocol to be implemented. With random oracles, which are usually used by the most efficient protocols, such a large gap exists.

In more detail, the network communication complexity of our protocol is in $\Theta(\mathsf{poly}(\kappa) + |S_2| \cdot \lambda)$, where λ is the statistical security parameter and $|S_2|$ the cardinality of P_2's set. In particular, it is independent of the set elements' size. The run-time complexity of our protocol is in $O(\mathsf{poly}(\kappa, \lambda, \ell)) \cdot (|S_1| + |S_2|))$, where ℓ is an upper bound for the set elements' size. Concretely, without having the implementation explicitly optimized for bandwidth, we measure 135MByte on sets of 2^{24} elements, which is roughly 64.1 bits per element, outperforming [92] at 300 bits per element for the same setting.

We achieve this performance by using TEEs, which are a powerful tool to guarantee the correct and secure execution of a computation. In our protocol, we use them i) to locally extract a corrupted party's input, allowing the network communication to be compressed and ii) to force honest behavior without having to rely on e.g. zero-knowledge proofs.

A main advantage of our approach is that we do not require full trust in the security of the used TEEs. Our main requirement is the correct attestation of an execution, together with

the existence of secure implementations for very simple cryptographic building blocks like key exchange, deterministic encryption and signature schemes, including secure handling of the keys.

Under these assumptions, our protocol can deal with the following cases:

- The TEE leaks all secrets to the host, except for locally performed simple cryptographic operations. This captures a setting where side channels for the general execution exist, but a limited set of operations can still be performed securely, e.g. because they have been implemented in a way that addresses platform-specific side-channels.
- The TEE leaks all secrets to the vendor or another party not participating in the protocol execution, including secrets of locally performed cryptographic operations. Intuitively, this captures the setting of a manufacturer or a government agency introducing backdoors to TEEs. (We call such enclaves semi-honest or passively corrupted.)

To the best of our knowledge, we are the first to consider TEEs with a realistic security model for the task of PSI. In any case, we can protect the private inputs of honest parties P_1 and P_2 from the TEEs. Surprisingly, we can also protect the size of the intersection from passively corrupted TEEs, which is not guaranteed by naïve constructions using TEEs or some server-aided PSI protocols.

Our main technical tool consists of a deterministic secret-key encryption scheme to hide a party's inputs from the TEEs and the other party, while still allowing efficient comparisons on ciphertexts. Using an appropriate operation mode, a deterministic encryption scheme can be constructed very efficiently using an arbitrary block cipher such as AES. In particular, this allows fast implementations when accelerated AES is available, e.g. via the widely-available AES-NI instruction set [52].

To reduce the communication complexity, we employ universal hashing. A key insight is that this hashing must be performed by the TEE in order to allow extraction.

In order to formally capture the properties of semi-honest or almost-transparent TEEs, we extend the model of Pass, Shi, and Tramèr [88]. Using this extended functionality, we prove the security of our protocol in the UC framework [23, 8].

To demonstrate the efficiency of our approach, we also provide an implementation of our protocol using Intel SGX [37] as TEEs.

2.1.3. Related Work

Private set intersection has been intensively studied during the previous decades, considering different aspects such as optimizing intersections between two parties versus considering arbitrary many parties, having a server that does not collude with the parties aide in the computation versus only having the parties take part or semi-honest versus malicious security.

We focus the examination of existing PSI protocols on three other publications as they consider a similar setting compared to our work, are fairly recent and highly efficient.

[90] introduces a new data structure called PaXoS (a probe-and-XOR of strings) from which the authors build a PSI protocol. The protocol has both a semi-honest and a malicious variant with only a small performance difference. [47] generalizes the idea of encoding one input set into a data structure, and improves the failure probability of such an encoding. The authors use

amplification techniques to ensure a low failure probability while simultaneously improving the runtime by 20-40%.

[92] further drastically improves the OKVS (oblivious key value store) constructions by using VOLE (vector oblivious linear evaluation) as the main primitive. Using similar techniques as the other protocols, this work also has a semi-honest and malicious variant with only a relatively small performance difference.

In order to achieve composable security and not resorting to random oracles, we assume trusted hardware that may have side-channels or be corrupted semi-honestly as a compromise between having no trusted hardware at all and performing the whole computation blindly trusting the hardware. Such a use case for trusted hardware was also considered by Lu et al. [76]. In their work, the authors use SGX enclaves to generate correlated randomness like randomized oblivious transfer tuples to use in generic stand-alone multi-party computation protocols. Their security model assumes trusted hardware, like an independent server in server-aided computation, that can only be corrupted independently of the parties that execute the protocol. If the trusted hardware in [76] is passively corrupted, the setting is comparable to our model of semi-honest enclaves (see section 2.2.3).However, our achieved security notion is stronger, featuring universal composability.

2.1.4. Insecurity of Previous Implementations

When giving a simulation-based proof of security, the simulator must be able to faithfully simulate the execution by what it learns from the ideal functionality. Often, protocols are given in pseudocode only, using mathematical abstractions such as sets for the protocol parties' inputs. When implementing the protocol, these abstractions have to be instantiated using data structures, e.g. by lists or arrays for variables that are typed as sets in the pseudocode.

A problem arises when these data structures have a different semantic. In lists or arrays, elements are ordered, whereas sets may be unordered. This may lead to subtle problems such as the implementation allowing to distinguish if an honest party's input is $\{1, 2\}$ versus $\{2, 3\}$, even if the intersection is $\{2\}$ in both cases, due to the position of the matching element. If UC security is considered, this is a problem, as the simulator only learns the intersection $\{2\}$, while the environment knows the input $\{1, 2\}$ resp. $\{2, 3\}$.

Two implementations [90, 92] exhibit this problem, allowing to learn something about the input of the party that does not learn the intersection, because the used data structures are ordered (see subsection 2.4.2). Due to the use of (unordered) sets in the protocol description, this is not accounted for in the proof. As the simulator is not given this leakage, the simulators stated (implicitly) in [90, 92] cannot correctly simulate in presence of this leakage. While the proof of security is arguably sound, a different and insecure protocol is implemented. While even more severe, this problem due to implementing a different protocol than the analyzed one is similar to the one that exists when random oracles are replaced by hash functions (see Use of Random Oracles in Previous Protocols in section 2.1).

In order to fix the vulnerability, we propose the implementation to be modified to hide the order of the input, e.g. by (pseudo)randomly permuting it or possibly by sorting the hashes or ciphertexts of the input, which also leads to a (pseudo)random permutation. Unfortunately, the price to pay for this solution is a decrease in efficiency, as sorting and permuting are computationally expensive.

We believe that this issue highlights a general problem with respect to the usability of cryptographic protocols: Many important assumptions are implicit (and thus overlooked later on, e.g. for the implementation), and today's protocols are increasingly complex in order to achieve the best performance or security guarantees, making their security hard to analyze.

2.1.5. Outline

After the preliminaries in section 1.3, we present our models for semi-trusted TEEs in section 2.2. Using these ideal functionalities, we give our construction for efficient one-sided two-party private set intersection in section 2.3. Then, we present our implementation and evaluate its performance in section 2.4. Finally, we present further applications of our model in section 2.5. The security proof can be found in section 2.3.2 and a short introduction into UC security in section 1.3.2.

2.2. Transparent Enclaves with Secure Operations

A secure enclave or trusted execution environment (TEE) a is a part of a system isolated from any interference of other parts of the platform (e.g. by the operating system or other applications running on the same system), promising the secure execution of arbitrary user-defined programs.

Remote Attestation. The real power of enclaves comes in the form of so-called remote attestation, which can be used to attest for another system that a specific output was computed by an enclave running a specific program. Intuitively, a remote attestation can be viewed as a signature of an enclave's output together with the enclave's program code using a secret key that is only available to the enclave. Often, e.g. in the case of Intel SGX, the verification key for such a signature would be globally known and shared between all protocols using enclaves, and as such a remote attestation would be verifiable by anyone.

Remote attestation can be used together with a key exchange to establish a secure channel into an enclave: The enclave performs one side of the key exchange and outputs the result with an attestation. The client verifies the attestation and completes the key exchange. The attestation confirms to the client that the other side of the key exchange was executed by an enclave running the specified program, and, if the enclave program does not leak the secret, no one outside will have access to the exchanged key. Additionally, the client can embed a verification key into the enclave's program and sign their part of the key exchange to ensure that the enclave will only accept input from them. After the key exchange has finished, the client can then send its inputs, encrypted with the exchanged key using e.g. an IND-CCA-secure encryption scheme, to the enclave.

Computing Arbitrary Functionalities. TEEs with remote attestation seem to solve secure computation problems very efficiently: The parties agree on a functionality and a system that they will trust to run the enclave. Every party providing input will perform a key exchange with the enclave and send their input as described above. Having all parties embed a verification key into the enclave's program prevents the entity managing the trusted hardware to mix up the parties'

inputs or swap them with their own. The enclave will perform the desired computation and uses the same secure channels to send each party their output.[2]

Side-Channels and Vulnerabilities. However, trusting the enclave completely with the inputs might not be desirable: First of all, today's enclaves like Intel SGX may have side-channels by design, e.g. in the form of allowing the observation of memory access patterns, which may depend on secrets stored inside the enclave.

Furthermore, the enclave might be subject to vulnerabilities. For Intel SGX, there was a series of (now patched) attacks on enclaves that range from extracting an enclave's inner state, injecting computation faults into an enclave's program or even extracting the attestation key. This allows an attacker to forge attestations and thereby break the security of enclaves completely. Therefore, protocols should require as little trust as possible.

Formal Models. Given their usefulness, TEEs have seen wide application in cryptography, with several formal models existing (e.g. [9, 102, 88, 76]). As we are interested in universal composability, we focus on the work of Pass, Shi, and Tramèr [88].

Often having one verification key for the remote attestation for many enclaves, e.g. as in the case of Intel SGX, a natural choice is to model TEEs in a framework like the Generalized Universal Composability (GUC) framework [24], which is the framework used by Pass, Shi, and Tramèr [88]. Following a recent result [8], global ideal functionalities can also be captured within the standard UC framework, which has been done for the ideal functionality of Pass, Shi, and Tramèr [88] in [14].

The ideal functionality of [88], called $\mathcal{G}_{att}$, models TEEs without any kind of side-channels. $\mathcal{G}_{att}$ uses a common signing key pair for signing remote attestations, where everyone can obtain the verification key. Further, $\mathcal{G}_{att}$ is parameterized with a list of parties that may install enclaves with arbitrary programs, called the registry. Once the installation has been performed, the installing party receives a handle for the corresponding enclave instance. A second interface of the ideal functionality allows parties to resume the execution of their own enclaves by providing input and obtain the program's output together with a signature on the session identifier, the enclave handle, the program's code and the program's output. This signature models remote attestation.

Unfortunately, the ideal functionality $\mathcal{G}_{att}$ is very optimistic in the sense that it considers the TEE to be completely trusted and does not model any side-channels—neither those inherent in the program that is to be executed, nor those present e.g. in Intel SGX due to memory access patterns.

To alleviate this, Pass, Shi, and Tramèr [88] also propose a variant of $\mathcal{G}_{att}$ to model so-called transparent enclaves, first introduced by Tramèr et al. [102]. For transparent enclaves, it is assumed that the attestation of the enclave system is secure (i.e. there will only be attestations of correct enclave outputs with the program that produces them), but that, at the same time, the enclave's owner has access to all of the enclave's randomness. In such a model, performing a

[2] Depending on the model considered, this approach may not yield provable security for technical reasons. For example the model of Pass, Shi, and Tramèr [88] provably requires two TEEs for any two-party functionality. However, as this is outside the scope of our paper, we will not further discuss it and refer the interested reader to [88].

key exchange with the enclave is useless, as the enclave owner would learn the secret key and could subsequently decrypt ciphertexts containing secret inputs. Here, leaking the randomness seems to capture many, if not all, side-channels.

Interestingly, transparent enclaves are still useful, as they can be used as very efficient passive-security-to-active-security compilers. In particular, very efficient protocols for commitments and zero-knowledge proofs are presented in [88].

A More Realistic Model. Given that Intel SGX's remote attestation is (in part) performed by a so-called quoting enclave provisioned by Intel that has access to the attestation key, but is not fundamentally different from any other enclave running on that system [37], transparent enclaves seem to be overly pessimistic. Of course, Intel has spent significant effort in hardening the code of the attestation enclave e.g. by using side-channel-free algorithms or placing memory barriers after security checks to prevent speculative execution, but the same argument might hold for careful implementations of cryptographic primitives offered by the trusted cryptographic library provided by the Intel SGX SDK, which can be used by any user-defined enclave program.

This observation suggests that there is a practically motivated and natural middle ground between fully trusted enclaves and transparent enclaves where additional cryptographic operations are possible in a secure way, whereas the rest of the execution is subject to side-channels (unless countermeasures are taken).

To this end, we extend the ideal functionality for transparent enclaves to additionally allow secure key exchange and symmetric encryption, in total analogy to the digital signatures in [88]. This choice is motivated by what operations are assumed to secure for Intel SGX, given appropriate side-channel-free implementations. However, we stress that this list is not final. Indeed, we envision further variants of our ideal functionality depending on which cryptographic primitives are needed and can plausibly be implemented in a secure way.

Adding these additional secure operations comes, however, with a caveat: Only leaking the enclave's randomness does no longer lead to a meaningful model. In the original model, transparent enclaves could not keep a secret from the host because the host could extract the key exchange by learning the randomness. In our model, key exchange between enclaves is possible. Consider the following protocol between parties P_1 and P_2 with enclaves E_1 and E_2 respectively:

1. E_1 and E_2 exchange a key k in a side-channel-free manner.
2. P_1 sends its secret x to its enclave E_1.
3. E_1 encrypts x using k in a side-channel-free manner to the ciphertext c and sends c to E_2 (via P_1 and P_2).
4. E_2 decrypts c to x using k in a side-channel-free manner and evaluates a deterministic function f that is affected by side-channels, e.g. in the form of memory access patterns, on x and halts without output.

Clearly, P_2 should learn x in the above example. In the original model for transparent enclaves, this would be the case as P_2 would learn k via the leaked randomness, could decrypt c and obtain x. If not for this leakage, P_2 could not learn x, as the computation of f is deterministic, leading to no leakage according to the model.

In order to address this problem, we augment our ideal functionality to not only leak randomness, but also i) the enclave memory as well as ii) the output of secure operations, e.g. released keys. This leads again to a meaningful model.

By explicitly distinguishing between secure operations and a side-channel-affected general execution, the requirements of an implementation become explicit.

2.2.1. Ideal Functionality With Secure Operations

We start with a definition of secure TEEs, i.e. TEEs without side-channels or leakage. This definition is based on the definition of Pass, Shi, and Tramèr [88] (with a fix from [14]), with the following differences:

1. In the original definition, the functionality is parameterized with a list of parties reg (the registry) able to install programs. Intuitively, this captures that only a subset of the protocol parties may have a TEE. In particular, the adversary (resp. simulator) is unable to install a program if all parties in the registry are honest. This leads to technical problems (see [88] for details). We have chosen to allow all parties and the adversary to create TEEs by removing reg. We believe that this is plausible in our practice-oriented setting due to the wide availability of TEEs such as Intel SGX.
2. We introduce a number of secure enclave operations, allowing to e.g. exchange keys between two enclaves or to perform deterministic encryption. While such secure enclave operations are not necessary if the TEEs are fully trusted, they are useful if we assume side-channels during the execution of enclave programs (see definitions 6 and 7). Note that in the original definition of transparent enclaves [102], the signature scheme used for the attestation signatures is also modeled to keep its security. If this were not the case, all security guarantees would be lost as an adversary could completely impersonate a TEE.

Definition 5 (Ideal Functionality $\mathcal{G}_{att}[\mathbf{Sig}, \mathbf{SKE}, \kappa]$)
// initialization:
On initialize: $(\mathrm{mpk}, \mathrm{msk}) \leftarrow \mathit{Sig.Gen}(1^\kappa), T = \emptyset, S = \emptyset$.

// public query interface
On receive* `getpk` from some P: Send mpk to P.

Secure enclave operations

On receive* $(\texttt{kex}, \mathrm{eid}_2)$ from a TEE with EID eid_1:

- Sample $k \xleftarrow{\$} \{0,1\}^\kappa$ and $\mathsf{hdl} \xleftarrow{\$} \{0,1\}^\kappa$.
- For $i \in \{1,2\}$, send $(\texttt{key-exchange}, \mathrm{eid}_1, \mathrm{eid}_2, \mathrm{eid}_i)$ to the adversary. Upon confirmation, set $S[\mathrm{eid}_i, \mathsf{hdl}] = k$ and output $(\mathrm{eid}_1, \mathrm{eid}_2, \mathsf{hdl})$ to the TEE with eid_i.

On receive* $(\texttt{ske.gen})$ from a TEE with EID eid:

- Let $k \leftarrow \mathit{SKE.Gen}(1^\kappa)$.
- Sample a random handle $\mathsf{hdl} \xleftarrow{\$} \{0,1\}^\kappa$, set $S[\mathrm{eid}, \mathsf{hdl}] = k$ and return hdl.

On receive* $(\texttt{ske.enc}, \mathsf{hdl}, m)$ from a TEE with EID eid:

- If there is no entry $S[\text{eid}, \mathsf{hdl}]$, abort. Otherwise, return $\mathit{SKE.Enc}(S[\text{eid}, \mathsf{hdl}], m)$.

On receive* $(\texttt{ske.dec}, \mathsf{hdl}, c)$ from a TEE with EID eid:

- If there is no entry $S[\text{eid}, \mathsf{hdl}]$, abort. Otherwise, return $\mathit{SKE.Dec}(S[\text{eid}, \mathsf{hdl}], c)$.

On receive* $(\texttt{release-key}, \mathsf{hdl})$ from a TEE with EID eid:

- If there is no entry $S[\text{eid}, \mathsf{hdl}]$, abort. Otherwise, return $S[\text{eid}, \mathsf{hdl}]$.

Normal enclave operations

// local interface — install an enclave:
On receive* $(\texttt{install}, \text{idx}, \mathsf{prog})$ from some ITI μ where μ is either a protocol party or the adversary:

- If P is honest, assert idx = sid.
- Generate nonce eid $\xleftarrow{\$} \{0,1\}^\kappa$, store $T[\text{eid}, \mu] = (\text{idx}, \mathsf{prog}, \vec{0})$, send eid to μ.

//local interface — resume an enclave
On receive* $(\texttt{resume}, \text{eid}, \mathsf{inp})$ from some ITI where μ is either a protocol party or the adversary:

- Let $(\text{idx}, \mathsf{prog}, \mathsf{mem}) = T[\text{eid}, \mu]$, abort if not found.
- Let $(\mathsf{outp}, \mathsf{mem}') = \mathsf{prog}(\mathsf{inp}, \mathsf{mem})$, update $T[\text{eid}, \mu] = (\text{idx}, \mathsf{prog}, \mathsf{mem}')$.
- Let $\sigma = \mathit{Sig.Sign}(\text{msk}, (\text{idx}, \text{eid}, \mathsf{prog}, \mathsf{outp}))$ and send (outp, σ) to μ.

Remark 1 For the sake of an easier description, the key exchange between enclaves is modeled in an ideal way similar to definition 3. Alternatively, we could have incorporated an explicit protocol such as the signed Diffie-Hellman key exchange.

Remark 2 Sometimes, we need to be able to iteratively perform secure enclave operations without leaking intermediate results. For example, we might want to encrypt a message m under a key with handle hdl and then encrypt the resulting ciphertext with a key with a different handle hdl' in a way that the ciphertext resulting from the encryption with hdl is not leaked.

To this end, we can naturally augment the secure enclave operations to optionally return handles only and to accept them as inputs instead of real messages. The `release-key` operation could be generalized to work on arbitrary handles. However, for the sake of better readability, we have chosen to not to include these additional modes of operation in the formal description of the model and our protocols.

Following this basic definition, we define two variants of TEEs that capture side-channels resp. semi-honest TEEs leaking all secrets to the adversary.

2.2.2. Almost-transparent Enclaves

We now consider a variant of the functionality in definition 5 where all normal enclave operations are affected by side-channels. Per the discussions above, it is insufficient to only leak the used randomness in the setting we consider. Instead, we also leak the pre-computation memory. Using this information, the computation can be fully reproduced, including its timing (which is not modeled), inputs and outputs as well as memory access patterns.

Definition 6 (Almost-transparent Enclave $\hat{\mathcal{G}}_{att}$) $\hat{\mathcal{G}}_{att}$ is defined identically to $\mathcal{G}_{att}$, except that besides outputting the pair (outp, σ) to the caller upon the `resume` entry point, it also leaks to the caller

- all random bits internally generated during the computation,
- the enclave memory mem pre-execution and
- the output of secure operations.

Note that this definition, adapted from [88], does not include randomness or memory used by secure enclave operations or the randomness used to generate the signing key.

2.2.3. Semi-honest Enclaves

Apart from side-channels that leak information to a TEE's host, we are also interested in the setting where the TEE is passively corrupted and leaks all information about the computation. In contrast to the leakage due to side-channels, this leakage could be to a third party not participating in the protocol execution, e.g. the manufacturer or a government agency. The fear of such leakage is plausible because many of today's TEE implementations are very complex, closed-source and could possibly interact with other hardware building blocks, e.g. the network interface, without knowledge of the operating system (OS). On Intel systems, out-of-band network communication is often supported by the chipset [67]. This adversarial model has been previously considered in [76], albeit in a much simpler model without composability.

Definition 7 (Semi-honest Enclave $\tilde{\mathcal{G}}_{att}$) $\tilde{\mathcal{G}}_{att}$ is defined identically to $\mathcal{G}_{att}$, except that for each activation, a tuple $(\mathsf{inp}, \mathsf{outp}, r)$ is stored, where inp denotes the functionality's input, outp the output (if applicable, $\perp$ otherwise) and r the randomness used during the activation, including secure operations. The adversary may retrieve all previously stored and new tuples by sending a special message `leak` to $\tilde{\mathcal{G}}_{att}$.

In contrast to transparent enclaves, semi-honest enclaves leak all inputs, outputs and randomness, including enclave operations and the randomness used for the signature key generation.

With semi-honest enclaves, one cannot simply execute a passively secure protocol fully inside the TEEs anymore and hope to achieve meaningful security.

Looking ahead, we find that our construction for PSI is secure even in the presence of semi-honest TEEs, as long as no protocol party is corrupted. This captures a setting where the TEE leaks everything to e.g. the manufacturer or a government agency, but does not cooperate with the parties using the TEEs.

2.2.4. Possible Realizations of Our Model

Our model of TEEs with secure and side-channel-free cryptographic operations is motivated by a number of existing systems:

- Enclaves that are free of side-channels are trivially able to perform cryptographic operations in a secure manner, given appropriate implementations.
- Intel SGX needs to perform a number of cryptographic operations in a secure manner for attestation [37]. In particular, this includes i) message authentication codes (MACs) and ii) digital signatures. Also, the trusted cryptographic library distributed by Intel as part of the SGX SDK contains implementations of i) key exchange and ii) secret-key encryption. Given the prevalence of SGX vulnerabilities, realizing secure enclave programs is indeed a difficult task. However, there are "best practices" provided by Intel [59, Protection from Side-Channel Attacks]. Additionally using memory barriers, constant-time code and data-oblivious implementations may be helpful to avoid side-channel leakage.

  We also note that one needs to consider the adversarial model of the used TEE. For example, advanced attacks using fault injection or power attacks may (or may not) invalidate the adversarial model altogether, rendering the assumption (that certain operations can be performed securely) wrong. In the case of SGX, mitigations for vulnerabilities covered by the adversarial model are available and are checkable via remote attestation.
- Intel SGX could also be combined with different technologies for trusted execution, e.g. Intel Trusted Execution Technology (TXT), to perform cryptographic operations. Intel TXT has previously been used to securely perform cryptographic operations [79].
- Many mobile devices that feature enclaves such as ARM TrustZone have dedicated secure hardware for cryptographic operations, e.g. recent Apple iPhones [6] or Google Pixel smartphones [80].
- Some IBM systems have a feature called IBM Secure Execution [58], which provides Linux-based secure enclaves. These could be combined with a dedicated hardware security module (HSM), which are also available from IBM.
- Secure enclaves are also supported by the open-source instruction set RISC-V [98]. If necessary, this instruction set could possibly be extended to be able to perform basic cryptographic operations in hardware.

2.3. One-sided Private Set Intersection

Before describing our protocol, we state the ideal functionality we realize. $\mathcal{F}_{\text{oPSI}}$, the ideal functionality for one-sided PSI, takes inputs S_1 and S_2 from P_1 resp. P_2 and computes the intersection $S = S_1 \cap S_2$, which is lexicographically sorted and returned to P_1. The lexicographical sorting is employed in order to avoid ambiguity and the introduction of leakage of the private inputs depending on the result's sorting. See section 2.1.4 for examples in previous protocols.

Definition 8 (Ideal Functionality $\mathcal{F}_{\text{oPSI}}$) Parameterized by a security parameter κ and two parties P_1 and P_2. The input of each party consists of a set with elements from $\{0,1\}^\kappa$.

- On input $(\texttt{input}, \text{sid}, S_1)$ from P_1, store S_1 and send $(\texttt{input}_1, \text{sid}, |S_1|)$ to the adversary.

- On input $(\texttt{input}, \text{sid}, S_2)$ from P_2, store S_2 and send $(\texttt{input}_2, \text{sid}, |S_2|)$ to the adversary.
- When having received input from both parties, compute the intersection $S = S_1 \cap S_2$ and sort S lexicographically. Also, send $(\texttt{size}, \text{sid}, |S|)$ to the adversary.
- Generate a private delayed output $(\texttt{result}, \text{sid}, S)$ for P_1.

Remark 3 In definition 8, the adversary always learns $|S_1|$, $|S_2|$ and $|S|$, regardless of which parties are corrupted. More fine-grained leakage can be modeled (and realized by our protocol) at the cost of a more complex description of $\mathcal{F}_{\text{oPSI}}$.

Remark 4 In $\mathcal{F}_{\text{oPSI}}$, only the party P_1 obtains the intersection result. If the considered universe U is small enough (e.g. when U consists of telephone numbers), a malicious party P_1 could use the whole universe U as its input and would then learn the input of P_2. Under notions of PSI where both parties learn the output, this could possibly be detected heuristically by P_2 after it has learned the result. In the case of one-sided PSI, this is not possible.

If protection against such an attack strategy is necessary, $\mathcal{F}_{\text{oPSI}}$ can be easily modified in a number of ways to e.g. i) limit the input size of either or both parties, ii) tell P_2 the size of P_1's input or to iii) only output the result to P_1 if a certain threshold for the intersection's size is met or kept.

Looking ahead, our construction could be easily adapted to the first two points while retaining its security properties. For the last point, an easy modification is possible that would, however, allow the TEEs to learn the intersection's size, which we currently prevent.

2.3.1. Our Protocol

We now present our construction π_{oPSI}. We assume the existence of authenticated communication and consider adversaries that statically corrupt protocol parties, i.e. at the very beginning of the execution.

The protocol description is split into three parts: The actual protocol executed by P_1 and P_2, together with the programs executed by each party's TEE.

Construction 1 (Protocol π_{oPSI}) Let Sig be a signature scheme and $\mathsf{SKE}_{\mathsf{detCPA}}$ a deterministic secret-key encryption scheme. Let $\kappa \in \mathbb{N}$ be a security parameter. Let $\mathcal{G}_{att} = \mathcal{G}_{att}[\mathit{Sig}, \mathsf{SKE}_{\mathsf{detCPA}}, \kappa]$.

Program prog_1 **of** P_1**'s enclave:**

1. Receive $(\text{mpk}, \text{eid}_1, \text{eid}_2, \text{prog}_2)$ as first input.
2. Execute $(\texttt{key-exchange}, \text{eid}_2)$. Eventually, receive a message $(\text{eid}_1', \text{eid}_2', \mathsf{hdl})$. Store hdl if $\text{eid}_1' = \text{eid}_1$ and $\text{eid}_2' = \text{eid}_2$.
3. On input $(\texttt{input}, \text{sid}, C_1)$:
   - Set $C_1' = \{c_i' \mid c_i' = \texttt{ske.enc}(\mathsf{hdl}, c_i), c_i \in C_1\}$.
   - Output $(\texttt{ready}, \text{sid})$.
   - Upon the next activation, output (sid, C_1').
4. Ignore all further messages.

Program prog_2 **of** P_2**'s enclave:**

1. Receive $(\mathrm{mpk}, \mathrm{eid}_1, \mathrm{eid}_2, \mathsf{prog}_2)$ as first input.
2. Eventually receive a message $(\mathrm{eid}_1', \mathrm{eid}_2', \mathsf{hdl})$. Store hdl if $\mathrm{eid}_1' = \mathrm{eid}_1$ and $\mathrm{eid}_2' = \mathrm{eid}_2$.
3. On input $(\texttt{query}, \mathrm{sid})$, execute $(\texttt{release-key}, \mathsf{hdl})$ to obtain k_2 and output $(\texttt{key}, \mathrm{sid}, k_2)$.
4. On input $(\texttt{input}, \mathrm{sid}, C_2, \lambda)$, sample a UHF UHF from the set of UHFs with domain $\{0,1\}^{\mathsf{poly}(\kappa)}$, codomain $\{0,1\}^{\lambda}$ and super-polynomial image size in λ where $\mathsf{poly}(\kappa)$ is a polynomial denoting the length of the ciphertexts to be hashed. Compute $H = \{h_i \mid h_i = \mathit{UHF}(c_i), c_i \in C_2\}$ and output $(\texttt{output}, \mathrm{sid}, \mathit{UHF}, H)$.
5. Ignore all further messages.

Protocol:

If activated without the initial input, a party yields the execution until the input is received.

1. Each party P_i (eventually) receives input $(\texttt{input}, \mathrm{sid}, S_i)$ and is parameterized with a security parameter κ and a statistical security parameter λ.
2. P_1 and P_2 each send $(\texttt{getpk})$ to $\mathcal{G}_{att}$ to obtain the master public key mpk.
3. P_1 and P_2 call $\mathcal{F}_{\mathrm{KE}}$ twice, using SIDs sid||1 and sid||2 (with P_1 acting as initiator) to obtain keys k_1, k_3.
4. P_1 sends $(\texttt{install}, \mathrm{sid}, \mathsf{prog}_1)$ to $\mathcal{G}_{att}$, where prog_1 is the program for P_1's enclave in construction 1. It obtains eid_1 as answer and sends eid_1 to P_2.
5. Similarly, P_2 installs an enclave with program prog_2, obtains eid_2 and sends it to P_1.
6. P_1 sends $(\mathrm{mpk}, \mathrm{eid}_1, \mathrm{eid}_2, \mathsf{prog}_2)$ to its TEE with EID eid_1 via $\mathcal{G}_{att}$. Conversely, P_2 sends $(\mathrm{mpk}, \mathrm{eid}_1, \mathrm{eid}_2, \mathsf{prog}_1)$ to its TEE with EID eid_2.
7. Let $S_1 = \{s_1^1, \ldots, s_1^{n_1}\}$. For $i = 1, \ldots, n_1$, let $c_1^i = \mathsf{SKE}_{\mathsf{detCPA}}.\mathit{Enc}(k_1, s_1^i)$. P_1 sends $C_1 = \{c_1^1, \ldots c_1^{n_1}\}$ to the TEE with EID eid_1 and receives $(\texttt{ready}, \sigma_1)$ as answer, which it forwards to P_2.
8. On receiving $(\texttt{ready}, \sigma_1)$ from P_1, P_2 verifies the signature σ_1 and sends, upon successful verification, $(\texttt{query})$ to its TEE with EID eid_2 to obtain k_2. Let $S_2 = \{s_2^1, \ldots, s_2^{n_2}\}$. Then, P_2 computes $c_2^i = \mathsf{SKE}_{\mathsf{detCPA}}.\mathit{Enc}(k_3, \mathsf{SKE}_{\mathsf{detCPA}}.\mathit{Enc}(k_2, \mathsf{SKE}_{\mathsf{detCPA}}.\mathit{Enc}(k_1, s_2^i)))$ for $i = 1, \ldots, n_2$, sorts $C_2 = \{c_2^1, \ldots, c_2^{n_2}\}$ lexicographically and sends it, together with the statistical security parameter λ, to the TEE with eid_2. It receives $(\texttt{output}, \mathrm{sid}, \mathit{UHF}, H, \sigma_H)$, which it forwards to P_1.
9. On receiving $(\texttt{output}, \mathrm{sid}, \mathit{UHF}, H, \sigma_H)$, P_1 verifies the signature σ_H. Upon successful verification, it verifies that H is consistent with λ, it queries its TEE with EID eid_1 to obtain C_1'. Then, it computes $S = \{s_1^j \mid s_1^j \in S_1 \wedge \mathit{UHF}(\mathsf{SKE}_{\mathsf{detCPA}}.\mathit{Enc}(k_3, c_j')) \in H, c_j' \in C_1'\}$, sorts S lexicographically and outputs $(\texttt{result}, \mathrm{sid}, S)$.

In order to prove the security of our protocol, it is crucial that the environment is not able to access $\mathcal{G}_{att}$ (and its variants) arbitrarily. In particular, the environment may access $\mathcal{G}_{att}$ only with identities of 1. corrupted parties (and thus through the adversary) or 2. (honest) parties with a session ID that does not belong to an actual session in the execution. The first criterion is necessary to achieve a meaningful security notion, while the last criterion prevents the environment from accessing $\mathcal{G}_{att}$ with identities belonging to the test session or challenge session. This is in line with what is assumed by prior work, in particular in [88, 14] For a discussion on the formalism, see [8, 22].

We formally define the following identity bound ξ.

Definition 9 (Identity Bound) Let eid = $(\mu, \text{sid}||\text{pid})$ be an extended identity. Then, $\xi(\text{eid}) = 1$ if and only if

- the party with extended ID eid is corrupted or
- sid is not the session ID of an ITI existing in the current execution.

With this definition at hand, we are ready to state our main theorem.

Theorem 1 Let $\hat{\pi}_{\text{oPSI}} = \pi_{\text{oPSI}}^{\mathcal{G}_{att} \to \hat{\mathcal{G}}_{att}}$ be the protocol where $\mathcal{G}_{att}$ is replaced with $\hat{\mathcal{G}}_{att}$ and let $\tilde{\pi}_{\text{oPSI}} = \pi_{\text{oPSI}}^{\mathcal{G}_{att} \to \tilde{\mathcal{G}}_{att}}$ be the protocol where $\mathcal{G}_{att}$ is replaced with $\tilde{\mathcal{G}}_{att}$.

If Sig is an EUF-CMA-secure signature scheme, $\mathsf{SKE}_{\mathsf{detCPA}}$ is a det-LoR-CPA-secure secret-key encryption scheme, κ is the computational security parameter and $\lambda \in \Theta(\kappa)$ is the statistical security parameter, then $\hat{\pi}_{\text{oPSI}}$ ξ-UC-realizes $\mathcal{F}_{\text{oPSI}}$ in the presence of $\hat{\mathcal{G}}_{att}[\mathit{Sig}, \mathsf{SKE}_{\mathsf{detCPA}}, \kappa], \mathcal{F}_{\text{KE}}$ under static corruptions.

Under the same assumptions, $\tilde{\pi}_{\text{oPSI}}$ ξ-UC-realizes $\mathcal{F}_{\text{oPSI}}$ in the presence of $\tilde{\mathcal{G}}_{att}[\mathit{Sig}, \mathsf{SKE}_{\mathsf{detCPA}}, \kappa], \mathcal{F}_{\text{KE}}$ if the adversary does not corrupt any protocol party.

Informally, this means that π_{oPSI} is secure if a) the TEEs are almost-transparent, even if P_1 and / or P_2 are corrupted or b) if the TEEs are semi-honest and P_1 and P_2 are honest. Then, the TEEs do not learn the intersection size $|S|$.

We give a very short proof sketch.

Proof 1 (sketch) If P_1 is corrupted, the simulator must be able to extract the inputs of P_1 and provide them to $\mathcal{F}_{\text{oPSI}}$. Also, the simulator must simulate messages leading to the correct output for P_1, only knowing the intersection, but not P_2's elements that are not in the intersection.

In order to extract P_1's input, the simulator reads the ciphertexts P_1 sends to its TEE, which are encrypted with k_1. As the simulator simulates the honest party P_2, it knows k_1 and can decrypt the ciphertexts, learning P_1's input S_1.

Upon providing this input to $\mathcal{F}_{\text{oPSI}}$, the simulator learns the intersection S as well as $|S_2|$. For elements in the intersection, it can then prepare appropriate ciphertexts such that P_1 can compute the correct results. For elements not in the intersection (which the simulator does not know), it can create "dummy ciphertexts" to random values.

If P_2 is corrupted and P_1 is honest, the simulator's task is easier as it only has to extract P_2's input, which it does as described above.

If both parties are honest and the TEE is passively corrupted, the simulator learns $|S_1|$ and $|S_2|$ and can sample "dummy elements" for S_1 and S_2. Using these dummy elements, it can execute the protocol on behalf of the honest parties.

For the full proof, see section 2.3.2.

Remark 5 For the case of semi-honest corruptions of P_1 and P_2, two-sided private set intersection, i.e. where both P_1 and P_2 obtain the result, can be easily obtained by executing π_{oPSI} twice, once in each direction.

Remark 6 In π_{oPSI}, we encrypt the input to the TEEs. This prevents a passively corrupted TEE from learning e.g. a party's private input and the intersection. If one is willing to abandon these security guarantees, one could provide the TEE with the sets S_1 and S_2 in the clear and have it apply a user-defined function on the inputs, e.g. a perceptual hash algorithm. This way, our protocol can be adapted to different applications while retaining its efficiency and security in the presence of side-channels.

Let ℓ be a bound for the set elements' size[3]. It is easy to see that the network communication complexity is in $\Theta(\mathsf{poly}(\kappa) + |S_2| \cdot \lambda)$, assuming that i) the key exchange protocol has a communication complexity in $\Theta(\mathsf{poly}(\kappa))$, which holds for $\mathcal{F}_{\text{KE}}$ as well as e.g. for the Diffie-Hellman key exchange, ii) the length public keys and signatures of the signature scheme Sig can also bounded by a polynomial $\mathsf{poly}(\kappa)$ and iii) the description of the UHF UHF is bounded by a polynomial $\mathsf{poly}(\kappa)$. In particular, the communication complexity is linear in $|S_2|$ and independent of $|S_1|$ and ℓ.

The runtime complexity is in $O((|S_1|+|S_2|)\cdot\mathsf{poly}(\kappa,\ell))$, assuming that i) $\mathcal{F}_{\text{KE}}$ resp. its instantiation can be executed in $O(\mathsf{poly}(\kappa))$ steps, ii) the ciphertexts of plaintexts with length ℓ and keys with length in $O(\kappa)$ can be computed in $O(\mathsf{poly}(\kappa,\ell))$ steps and have length $O(\kappa+\ell)$, iii) the sorting can be performed in $O((\kappa+\ell)\cdot|S_2|)$ resp. $O(\lambda\cdot|S|)$ steps, iv) the UHF can be sampled and evaluated (once) in $O(\mathsf{poly}(\ell,\kappa,\lambda))$ steps and v) the signatures can be verified in $O(\mathsf{poly}(\kappa))$ steps. If $\ell \in O(\kappa)$, we obtain a runtime complexity of $O((|S_1| + |S_2|) \cdot \mathsf{poly}(\kappa))$.

Remark 7 Extending the above protocol to the multi-party setting is non-trivial. Let P_1 be the party that is supposed to receive the intersection and let $P_2, \ldots, P_n$ be the other parties. First, we observe that merely executing the protocol $n-1$ times, once between P_1 and each other party, is insecure in the sense that P_1 learns more than the intersection of all sets.

This problem also needs to prevented in a modified protocol: Currently, the party learning the intersection has to collect messages from each party which it can only process to compute the joint intersection, but not the individual ones. This seems to require a different protocol altogether and we leave the case of PSI with more than two parties in our model as future work.

2.3.2. Security Proof

In the following, we prove theorem 1, assuming that $\lambda \in \Theta(\kappa)$. To this end, we must prove that $\mathsf{M}[\pi_{\text{oPSI}}, \mathsf{IDEAL}(\mathcal{G})]$ ξ-UC-emulates $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_{\text{oPSI}}), \mathsf{IDEAL}(\mathcal{G})]$, where $\mathcal{G}$ is the global functionality under considertation and $\mathsf{IDEAL}(\mathcal{F})$ denotes the ideal protocol of the ideal functionality $\mathcal{F}$.

Proof 2 For the sake of an easier proof, we distinguish between different sets of honest and corrupted parties and state a simulator for each set. By combining the individual simulators, we obtain a universal simulator that is valid for all cases. We only consider simulators for the dummy adversary, which is complete (see section 1.3.2).

[3] In $\mathcal{F}_{\text{oPSI}}$ and π_{oPSI}, we assume that $\ell = \kappa$. However, this can be easily generalized.

Remark 8 $\mathcal{F}_{\text{oPSI}}$ (definition 8) always sends $|S_1|$, $|S_2|$ and $|S|$ to the adversary (see remark 3). By taking a closer look at the simulators given below, one can see that, depending on which parties are corrupted, not all leaked information is used for the simulation. In these cases, we could prove the security of our construction relative to an ideal functionality that does not leak the unused information. For the sake of better readability, we have chosen to not model this explicitly.

Both Parties Honest. When both parties of π_{oPSI} are honest, the enclaves are not transparent for the environment. We thus do not have to consider leakage, i.e. randomness, pre-execution memory or output of secure operations (see definition 6).

Definition 10 (Simulator $\mathcal{S}$, Both Parties Honest)

1. Behave like the dummy adversary for messages from $\hat{\mathcal{G}}_{att}$ and inputs from and outputs for corrupted parties.
2. Sample $k_3' \xleftarrow{\$} \{0,1\}^\kappa$.
3. Let $j \in \{1,2\}$ be the index of the party that first receives input. Let $n_j = |S_j|$ be the size of this party's input as reported by $\mathcal{F}_{\text{oPSI}}$.
4. Execute the protocol π_{oPSI} according to the current state, including the creation of TEEs.
5. Receive $n_{3-j} = |S_{3-j}|$ and $n = |S|$ from $\mathcal{F}_{\text{oPSI}}$. Set $S_2 = \{s_2^1, \ldots, s_2^{n_2}\}$ for pairwise distinct $s_2^i \xleftarrow{\$} \{0,1\}^\kappa$.
6. Continue the protocol execution, but use k_3' instead of k_3. When all messages have been delivered, i.e. when P_1 would accept in the real protocol, allow the output of $\mathcal{F}_{\text{oPSI}}$.

We show the indistinguishability of the following hybrids:

- H_0: The real execution with $\mathsf{M}[\pi_{\text{oPSI}}, \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$ and the dummy adversary $\mathcal{D}$.
- H_1: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_1), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$, where $\mathcal{F}_1$ is the ideal functionality that reports all inputs to the adversary and lets it determine all outputs. The simulator $\mathcal{S}_1$ works by executing π_{oPSI} for the honest parties, making outputs through $\mathcal{F}_1$.
- H_2: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_2), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$, where $\mathcal{F}_2$ is identical to $\mathcal{F}_1$. $\mathcal{S}_2$ is identical to $\mathcal{S}_1$, but calculates the result as $\mathcal{F}_{\text{oPSI}}$ would, i.e. using the parties' inputs reported by $\mathcal{F}_2$.
- H_3: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_3), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$, where $\mathcal{F}_3$ is identical to $\mathcal{F}_2$. $\mathcal{S}_3$ is identical to $\mathcal{S}_2$, but samples a new key k_3' and uses it instead of the k_3 exchanged in the protocol.
- H_4: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_4), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$, where $\mathcal{F}_4$ is identical to $\mathcal{F}_3$. $\mathcal{S}_4$ is identical to $\mathcal{S}_3$, but uses n_2 random distinct dummy inputs instead of S_2 when computing the messages of P_2.
- H_5: The ideal execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_{\text{oPSI}}), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$ and $\mathcal{S}$.

Claim 1 out_0 and out_1 are identically distributed.

Proof 3 As the changes between H_0 and H_1 are only syntactical and oblivious for the environment, the claim follows.

Claim 2 Let $H : \{h : \{0,1\}^{\mathsf{poly}(\kappa)} \to M_\lambda\}_{\kappa\in\mathbb{N},\lambda\in\mathbb{N}}$ be the family of universal hash functions with $1/M_\lambda = \mathsf{negl}(\lambda)$ for some negligible function negl and let $\mathit{UHF} \in H$. Then, out_1 and out_2 are statistically close in λ.

Proof 4 Unless the output of P_1 differs between the hybrids, out_1 and out_2 are identically distributed. In H_1, the output is computed according to the protocol (on the real inputs). In H_2, the output is computed relatively to the parties' inputs without considering possible collisions occurring in the protocol. This can lead to different results if and only if a collision in UHF occurs in H_1. As UHF is universal and was chosen after the input, we can bound the collision probability as follows.

Let $\mathrm{E}^i_{\mathrm{COL}}$ denote the event that the hash of the ciphertext of the i-th element c^i_1 (under k_1, k_2 and k_3) in S_1 creates a collision with any ciphertext of an element in S_2 (under k_1, k_2 and k_3), i.e. there exists a $s^2_j \in S_2$ such that $s^i_1 \neq s^j_2$ but $\mathit{UHF}(c^i_1) = \mathit{UHF}(c^j_2)$. (Due to the correctness of $\mathsf{SKE}_{\mathsf{detCPA}}$, it holds that $s^1_i \neq s^2_j \implies c^i_1 \neq c^i_2$.)

For fixed i, j, we have $\Pr[s^1_i \neq s^2_j \wedge \mathit{UHF}(c^i_1) = \mathit{UHF}(c^j_2)] \leq 1/M$, where M is the size of UHF's image. Using the union bound, we obtain $\Pr[\mathrm{E}^i_{\mathrm{COL}}] \leq |S_2|/M$. Let $\mathrm{E}_{\mathrm{COL}} = \mathrm{E}^1_{\mathrm{COL}} \cup \cdots \cup \mathrm{E}^{n_1}_{\mathrm{COL}}$. Using the union bound, we obtain $\Pr[\mathrm{E}_{\mathrm{COL}}] \leq (|S_1| \cdot |S_2|)/M$, which is negligible as $|S_1|$ and $|S_2|$ are polynomially bounded and M is super-polynomial.

It thus follows that out_1 and out_2 are statistically close in λ.

Claim 3 out_2 and out_3 are identically distributed.

Proof 5 As both parties are honest, the result of $\mathcal{F}_{\mathrm{KE}}$ is information-theoretically hidden from the environment. Thus, using a different key k'_3 does not change its view.

Claim 4 If $\mathsf{SKE}_{\mathsf{detCPA}}$ is det-LoR-CPA-secure, then out_3 and out_4 are computationally indistinguishable.

Proof 6 Suppose for the sake of contradiction that $|\Pr[\mathrm{out}_3 = 1] - \Pr[\mathrm{out}_4 = 1]| = \nu$ is non-negligible. Then, there exists an adversary $\mathcal{A}'$ against the det-LoR-CPA security of $\mathsf{SKE}_{\mathsf{detCPA}}$ with non-negligible advantage.

$\mathcal{A}'$ works as follows:

- On input $(1^\kappa, z)$, start an internal emulation of H_4 with input $(1^\kappa, z)$ for the environment.
- Internally emulate H_4, but create the ciphertexts with key k_3 as follows: For the dummy element $s^i_2 \in S_2$, let r^i_2 denote the corresponding real element of P_2's input. Let $R = \{r^1_2, \ldots, r^{n_2}_2\}$ and let $\mathrm{E}_{\mathrm{COL}}$ denote the event that $R \cap S_2 \neq \emptyset$. If $\mathrm{E}_{\mathrm{COL}}$ happens, output a uniformly random bit $b' \xleftarrow{\$} \{0,1\}$. Otherwise, prepare ciphertexts of s^i_2 and r^i_2 under k_1 and k_2, denoted by c_i and c'_i. Send (c_i, c'_i) to the encryption oracle and use the resulting ciphertext c^*_i as encryption of s_i resp. r_i. For elements of P_1, do not create the ciphertexts under k_3.

- Allow the output of P_1 in the internal simulation after $\mathcal{Z}$ has allowed the delivery of the final message by P_2.
- Output what $\mathcal{Z}$ outputs.

Due to the perfect correctness of $\mathsf{SKE}_{\mathsf{detCPA}}$ and the handling of $\mathrm{E}_{\mathrm{COL}}$, $\mathcal{A}'$ is a legal adversary for the det-LoR-CPA game.

If $\neg\mathrm{E}_{\mathrm{COL}}$ and the det-LoR-CPA game's choice bit b is 0, then $\mathcal{Z}$'s view is identically distributed to H_3. If $\neg\mathrm{E}_{\mathrm{COL}}$ and the choice bit is 1, then it is identically distributed to H_4. Thus, the advantage of $\mathcal{A}'$ in the det-LoR-CPA game is identical to $\mathcal{Z}$'s distinguishing advantage between H_3 and H_4, conditioned on $\neg\mathrm{E}_{\mathrm{COL}}$. In the case of $\mathrm{E}_{\mathrm{COL}}$, its advantage is 0. As $\Pr[\mathrm{E}_{\mathrm{COL}}]$ is negligible, the advantage of $\mathcal{A}'$ in the det-LoR-CPA game is $\nu \cdot (1 - \mathsf{negl})$ for some negligible function negl, which is non-negligible if ν is non-negligible, leading to a contradiction. It thus follows that $|\Pr[\mathrm{out}_3 = 1] - \Pr[\mathrm{out}_4 = 1]|$ is negligible.

Claim 5 out_4 and out_5 are identically distributed.

Proof 7 In H_4, the simulator $\mathcal{S}_4$ uses the real inputs of P_1. In contrast, the simulator $\mathcal{S}$ in H_5 uses appropriately distributed dummy values. However, the environment's view in H_4 is already independent of P_1's input. As its view remains identically distributed, it follows that out_4 and out_5 are identically distributed as well.

P_1 Corrupted. In the case of a corrupted P_1, the environment expects to see the randomness used by a TEE owned by a corrupted party as well as the pre-execution memory and the output of secure operations (if applicable). As the program prog_1 of P_1's TEE is deterministic, no randomness is used.

Definition 11 (Simulator $\mathcal{S}$, P_1 Corrupted)

1. Behave like the dummy adversary for messages from $\hat{\mathcal{G}}_{att}$ and inputs from and outputs for corrupted parties, except for the test session.
2. For P_1, always report the randomness from $\hat{\mathcal{G}}_{att}$, the pre-execution memory and the output of secure enclave operations.
3. Perform the preamble phase with P_1, learning k_1 and k_3.
4. Forward messages from P_1 to the TEE with EID eid_1.
5. When receiving $(\texttt{ready}, \sigma_1)$ from P_1:
   - If σ_1 is an invalid signature, halt.
   - If σ_1 is a valid signature but P_1 has not sent a message $(\texttt{input}, \ldots)$ to its TEE resulting in the output $(\texttt{ready}, \sigma')$, output a special symbol $\perp$.

- Otherwise, extract the input of P_1 as follows: Let C_1 denote the message sent from P_1 to the TEE with EID eid_1. Let S_1' denote the set that results from decrypting the ciphertexts in C_1 with k_1 (ciphertexts for which the decryption fails are ignored). Let S_1^* be the set that results from encrypting the elements in S_1' with k_1. Let S_1 be the set of all plaintexts that have ciphertexts in $S_1^* \cap C_1$ and send S_1 to $\mathcal{F}_{\text{oPSI}}$ on behalf of P_1. (We have to encrypt the decrypted elements again in order to protect from incorrectly created ciphertexts that decrypt correctly.)

6. Upon receiving $n_2 = |S_2|$, $n = |S|$ and S from $\mathcal{F}_{\text{oPSI}}$, sample distinct (pairwise different and distinct from P_1's extracted input) $s_2^1, \ldots, s_2^{n_2-n} \xleftarrow{\$} \{0,1\}^\kappa \setminus S_1$.
7. Continue the execution as P_2 would, using $S_2 = \{s_2^1, \ldots, s_2^{n_2-n}\} \cup S$ as input.

We consider the following hybrids:

- H_0: The real execution with $\mathsf{M}[\pi_{\text{oPSI}}, \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$ and $\mathcal{D}$.
- H_1: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_1), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$, where $\mathcal{F}_1$ is the ideal functionality that reports all inputs to the adversary and lets it determine all outputs. The simulator $\mathcal{S}_1$ works by executing π_{oPSI} for the honest parties, making outputs through $\mathcal{F}_1$.
- H_2: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_2), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$, where $\mathcal{F}_2$ is identical to $\mathcal{F}_1$. $\mathcal{S}_2$ is identical to $\mathcal{S}_1$, but determines the intersection by extracting P_1's ciphertexts and generates ciphertexts of dummy elements for elements of P_2 not in the intersection.
- H_3: The ideal execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_{\text{oPSI}}), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$ and $\mathcal{S}$.

Claim 6 out_0 and out_1 are identically distributed.

Proof 8 As the changes between H_0 and H_1 are only syntactical and oblivious for the environment, the claim follows.

Claim 7 If $\mathsf{SKE}_{\mathsf{detCPA}}$ is det-LoR-CPA-secure and perfectly correct and Sig is EUF-CMA-secure and perfectly correct, then out_1 and out_2 are computationally indistinguishable.

Proof 9 Let E_{FORGE} denote the event that the simulator outputs $\perp$. By definition, E_{FORGE} occurs if the environment has sent valid signature σ_1 without having received a valid signature σ' for the same message from the TEE. (Note that we only require EUF-CMA security for Sig. This does not rule out that the environment is able to derive a different valid signature $\sigma'' \neq \sigma'$ from σ', which is unproblematic.)

As Sig is EUF-CMA-secure, it follows that $\Pr[\text{E}_{\text{FORGE}}]$ is bounded by a negligible function $\mathsf{negl}_{\text{EUF-CMA}}$. As the reduction is straight-forward, we omit it.

If E_{FORGE} does not occur, the only difference between H_1 and H_2 are the plaintexts used to create the ciphertexts of P_2 for elements not in the intersection.

Let ν be the distinguishing advantage between H_1 and H_2 conditioned on $\neg\text{E}_{\text{FORGE}}$. If ν is non-negligible, we can construct an adversary $\mathcal{A}'$ against the det-LoR-CPA security of $\mathsf{SKE}_{\mathsf{detCPA}}$ with non-negligible advantage.

To this end, consider the following adversary $\mathcal{A}'$ against the det-LoR-CPA security of $\mathsf{SKE}_{\mathsf{detCPA}}$:

1. On input $(1^\kappa, z)$, start an internal emulation of H_1 with input $(1^\kappa, z)$ for the environment.
2. Let C_1 denote the ciphertext vector sent by P_1 to its TEE. Let S_1 denote the set of plaintexts for which the ciphertexts in C_1 are correct, i.e. $\{s_1^i \mid s_i = \mathsf{SKE}_{\mathsf{detCPA}}.\mathit{Dec}(k_1, c_1^i) \wedge \mathsf{SKE}_{\mathsf{detCPA}}.\mathit{Enc}(k_1, s_1^i) = c_1^i \wedge c_1^i \in C_1\}$.
3. Let S_2 denote the input of the honest party P_2. Let $S = S_1 \cap S_2$ denote the intersection.
4. Send (c_1^i, c_1^i) for $c_1^i \in C_1$ to the encryption oracle of the det-LoR-CPA game to obtain C_1'. Report it to $\mathcal{Z}$ as coming from the TEE (together with a correct signature).
5. Sample $|S_2| - |S|$ pairwise distinct random elements s_2^{i*} from $\{0,1\}^\kappa \setminus S_1$. Let S_2' denote the resulting set.
6. In order to obtain the ciphertexts of P_2, proceed as follows:
   - For elements inside the intersection, re-use the ciphertexts obtained in item 4.
   - For elements not in the intersection, Let $C_2 = \{c_2^1, \ldots, c_2^{n_2-n}\}$ denote the set of ciphertexts resulting from encrypting the elements in S_2' with k_1. Let $C_2' = \{c_2^{1,*}, \ldots, c_2^{n_2-n,*}\}$ denote the set of ciphertexts resulting from encrypting the elements of $S_2 \setminus S$ with k_1. For $i = 1, \ldots, n_2 - n$, send $(c_2^i, c_2^{i,*})$ to the encryption oracle.
7. Continue the execution using these ciphertexts.
8. Output what the environment outputs.

First, we note that $\Pr[\neg \mathrm{E}_{\mathrm{FORGE}}]$ is independent of the choice bit b used by the det-LoR-CPA experiment. Thus, $\Pr[\neg \mathrm{E}_{\mathrm{FORGE}}]$ is identical in the execution of H_1, H_2 and the reduction.

Conditioned on no signature forgery, it holds that if the choice bit in the det-LoR-CPA experiment is 0, then $\mathcal{Z}$'s view is identically distributed to H_1. If the choice bit is 1, then its view is identically distributed to H_2.

As such, an environment distinguishing between H_1 and H_2 with non-negligible probability v (conditioned on $\neg \mathrm{E}_{\mathrm{FORGE}}$) contradicts the security of $\mathsf{SKE}_{\mathsf{detCPA}}$.

All in all, we obtain $|\Pr[\mathrm{out}_1 = 1] - \Pr[\mathrm{out}_2 = 1]| \leq \mathsf{negl}_{\mathrm{EUF-CMA}} + \mathsf{negl}_{\mathrm{det-LoR-CPA}}$ for functions $\mathsf{negl}_{\mathrm{EUF-CMA}}$ and $\mathsf{negl}_{\mathrm{det-LoR-CPA}}$ bounding an adversary's advantage in the EUF-CMA resp. det-LoR-CPA game, which is negligible.

Claim 8 out_2 and out_3 are identically distributed.

Proof 10 As the changes between H_2 and H_3 are only syntactical and oblivious for the environment, the claim follows.

P_2 **Corrupted.** In this execution, the TEE is no longer deterministic as it samples a universal hash function.

Definition 12 (Simulator $\mathcal{S}$, P_2 Corrupted)

1. Behave like the dummy adversary for messages from $\hat{\mathcal{G}}_{att}$ and inputs from and outputs for corrupted parties, except for the test session.
2. For P_2, always report the randomness from $\hat{\mathcal{G}}_{att}$, the pre-execution memory and the output of secure enclave operations.
3. Perform the preamble phase honestly.
4. Receive $n_1 = |S_1|$ from $\mathcal{F}_{\text{oPSI}}$. Generate $|S_1|$ distinct dummy ciphertexts under k_1 and send them to the TEE with EID eid_1. Receive the answer $(\texttt{ready}, \sigma_1)$ and forward it to P_2.
5. Receive $(\texttt{ready}_2, \sigma_2)$ from P_2. If σ_2 is invalid, halt. If σ_2 is valid, but P_2 has not made the input to its TEE resulting in the signature for $\texttt{ready}_2$, output $\perp_1$. Otherwise, continue the protocol execution like P_1 would.
6. Receive $(\texttt{output}, \text{sid}, \mathit{UHF}, H, \sigma_H)$ from P_2. If σ_H is valid, but
   - P_2 has not made the input to its TEE resulting in a signature for H or
   - H has been computed incorrectly (relative to the provided UHF UHF and the set C_2 previously sent by P_2 to its TEE),

   output $\perp_2$. Otherwise, i.e. if everything is consistent and UHF is a UHF with statistical security parameter λ, extract P_2's input as follows: First, decrypt the ciphertexts (denoted by C_2) sent to the TEE with EID eid_2. Ignore ciphertexts that do not decrypt properly. Encrypt the resulting plaintexts under k_1, k_2 and k_3, resulting in C_2'. Let S_2 denote the extracted input comprised of elements that have ciphertexts in C_2 and C_2' (i.e. ciphertexts in C_2 that have been created correctly). Send S_2 to $\mathcal{F}_{\text{oPSI}}$ on behalf of P_2.
7. Allow the output of $\mathcal{F}_{\text{oPSI}}$ for P_1.

We consider the following hybrids:

- H_0: The real execution with $\mathsf{M}[\pi_{\text{oPSI}}, \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$ and $\mathcal{D}$.
- H_1: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_1), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$, where $\mathcal{F}_1$ is the ideal functionality that reports all inputs to the adversary and lets it determine all outputs. The simulator $\mathcal{S}_1$ works by executing π_{oPSI} for the honest parties, making outputs through $\mathcal{F}$.
- H_2: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_1), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$ where $\mathcal{F}_2 = \mathcal{F}_1$. $\mathcal{S}_2$ is identical to $\mathcal{S}_1$, but extracts the inputs of P_2 like $\mathcal{S}$.n
- H_3: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_1), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$ where $\mathcal{F}_3 = \mathcal{F}_1$. $\mathcal{S}_3$ is identical to $\mathcal{S}_4$, but uses dummy values for the input of P_1.
- H_4: The ideal execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_{\text{oPSI}}), \mathsf{IDEAL}(\hat{\mathcal{G}}_{att})]$ and $\mathcal{S}$.

Claim 9 out_0 and out_1 are identically distributed.

Proof 11 As the changes between H_0 and H_1 are only syntactical and oblivious for the environment, the claim follows.

Claim 10 If $\mathsf{SKE}_{\mathsf{detCPA}}$ is perfectly correct and Sig is EUF-CMA-secure and perfectly correct, then out_1 and out_2 are computationally indistinguishable.

Proof 12 Let $\mathrm{E}_{\mathrm{FORGE}}$ denote the event of a forgery of either signature sent by P_2, i.e. the simulator outputting $\perp_1$ or $\perp_2$. Due to the EUF-CMA security, the probability that P_2 presents a valid forged signature is negligible, i.e. $\Pr[\mathrm{E}_{\mathrm{FORGE}}] \leq \mathsf{negl}_{\mathrm{EUF-CMA}}$ for some negligible function $\mathsf{negl}_{\mathrm{EUF-CMA}}$ bounding an adversary's advantage in the EUF-CMA game. As the reduction is straight-forward, we omit it.

Conditioned on $\neg\mathrm{E}_{\mathrm{FORGE}}$, the only possibility for a difference to occur between H_1 and H_2 is a difference in the output, as its calculation is the only difference between the hybrids. For this, one of the following needs to happen:

- The simulator extracts invalid inputs for P_2.
- There is an element in S_1 that is not in the real intersection, but for which a collision with one of the hashes in H occurs.

As $\mathsf{SKE}_{\mathsf{detCPA}}$ is perfectly correct, the first case does not occur. We note that we cannot directly verify that the ciphertexts of P_2 have been created correctly. Due to the perfect correctness of $\mathsf{SKE}_{\mathsf{detCPA}}$, a ciphertext c that has been honestly created will always decrypt successfully. Conversely, for a ciphertext that decrypts successfully but is not in the image of Enc or for a ciphertext that c does not decrypt correctly, there will be no corresponding ciphertext created by the honest party. Thus, such an incorrect ciphertext cannot influence the calculation of the intersection unless a hash collision occurs.

Let $\mathrm{E}_{\mathrm{COL}}$ the event of a hash collision (regardless of whether the ciphertext has been created honestly or not). As UHF is a universal hash function with appropriate domain and codomain that was chosen after the inputs of P_2 are fixed (conditioned on no signature forgery), the probability of hash collisions is negligible, i.e. $\Pr[\mathrm{E}_{\mathrm{COL}}] \leq \mathsf{negl}_{\mathrm{UHF}}(\lambda)$ for some negligible function $\mathsf{negl}_{\mathrm{UHF}}$. For a detailed argument, see the proof of claim 2.

All in all, we have $|\Pr[\mathrm{out}_1 = 1] - \Pr[\mathrm{out}_2 = 1]| \leq \mathsf{negl}_{\mathrm{EUF-CMA}} + \mathsf{negl}_{\mathrm{UHF}}$, which is negligible.

Claim 11 out_2 and out_3 are identically distributed.

Proof 13 Even though the simulator uses different inputs for its message to the TEE with EID eid_1, the environment's view remains identically distributed as it does not see these messages. Also, the dummy values are not used to determine the result.

Claim 12 out_3 and out_4 are identically distributed.

Proof 14 As the changes between H_3 and H_4 are only syntactical and oblivious for the environment, the claim follows.

TEEs Passively Corrupted. This case is similar to the case that both parties are honest. Here, the adversary is additionally able to learn all messages sent from or to $\tilde{\mathcal{G}}_{att}$, as well as any randomness used by $\tilde{\mathcal{G}}_{att}$.

Definition 13 (Simulator $\mathcal{S}$, TEEs Passively Corrupted)

1. Behave like the dummy adversary for messages from $\tilde{\mathcal{G}}_{att}$ and inputs from and outputs for corrupted parties.
2. Answer `leak` queries by sending `leak` to $\tilde{\mathcal{G}}_{att}$ and reporting the result.
3. Sample $k'_1, k'_3 \xleftarrow{\$} \{0,1\}^\kappa$.
4. Let $j \in \{1,2\}$ be the index of the party that first receives input. Let $n_j = |S_j|$ be the size of this party's input as reported by $\mathcal{F}_{\text{oPSI}}$. Sample distinct (relative to other dummy elements) $s_j^1, \ldots, s_j^{n_j} \xleftarrow{\$} \{0,1\}^\kappa$ and set $S_j = \{s_j^1, \ldots, s_j^{n_j}\}$.
5. Execute the protocol π_{oPSI} according to the current state.
6. Receive $n_{3-j} = |S_{3-j}|$ and $n = |S|$ from $\mathcal{F}_{\text{oPSI}}$ and set $S_{3-j} = \{s_{3-j}^1, \ldots, s_{3-j}^{n_{3-j}}\}$ for distinct $s_{3-j}^i \xleftarrow{\$} \{0,1\}^\kappa$.
7. Continue the protocol execution, but use k'_1 instead of k_1 and k'_3 instead of k_3. When all messages have been delivered, i.e. when P_1 would accept in the real protocol, allow the output of $\mathcal{F}_{\text{oPSI}}$.

Remark 9 Even though the simulator learns $|S|$ from $\mathcal{F}_{\text{oPSI}}$, $|S|$ is not needed for the simulation. Thus, we can also prove security relative to a variant of $\mathcal{F}_{\text{oPSI}}$ that does not give $|S|$ to the adversary for the case that both parties are honest and the TEE is semi-honest.

We consider the following hybrids:

- H_0: The real execution with $\mathsf{M}[\pi_{\text{oPSI}}, \mathsf{IDEAL}(\tilde{\mathcal{G}}_{att})]$ and $\mathcal{D}$.
- H_1: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_1), \mathsf{IDEAL}(\tilde{\mathcal{G}}_{att})]$, where $\mathcal{F}_1$ is the ideal functionality that reports all inputs to the adversary and lets it determine all outputs. The simulator $\mathcal{S}_1$ works by executing π_{oPSI} for the honest parties, making outputs through $\mathcal{F}_1$.
- H_2: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_2), \mathsf{IDEAL}(\tilde{\mathcal{G}}_{att})]$, where $\mathcal{F}_2 = \mathcal{F}_1$. $\mathcal{S}_2$ is identical to $\mathcal{S}_1$, but uses a random key k'_1 instead of k_1.
- H_3: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_3), \mathsf{IDEAL}(\tilde{\mathcal{G}}_{att})]$, $\mathcal{F}_3 = \mathcal{F}_2$. $\mathcal{S}_3$ is identical to $\mathcal{S}_2$, but replaces ciphertexts of S_1 with ciphertexts of dummy values like $\mathcal{S}$.
- H_4: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_4), \mathsf{IDEAL}(\tilde{\mathcal{G}}_{att})]$, $\mathcal{F}_4 = \mathcal{F}_2$. $\mathcal{S}_4$ is identical to $\mathcal{S}_3$, but uses a random key k'_3 instead of k_3.
- H_5: Execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_5), \mathsf{IDEAL}(\tilde{\mathcal{G}}_{att})]$, $\mathcal{F}_5 = \mathcal{F}_4$. $\mathcal{S}_3$ is identical to $\mathcal{S}_2$, but replaces ciphertexts of S_2 with ciphertexts of dummy values like $\mathcal{S}$.
- H_6: The ideal execution with $\mathsf{M}[\mathsf{IDEAL}(\mathcal{F}_{\text{oPSI}}), \mathsf{IDEAL}(\tilde{\mathcal{G}}_{att})]$ and $\mathcal{S}$.

Claim 13 out_0 and out_1 are identically distributed.

Proof 15 As the changes between H_0 and H_1 are only syntactical and oblivious for the environment, the claim follows.

Claim 14 out_1 and out_2 are identically distributed.

For a proof, see the proof of the very similar claim 3.

Claim 15 If $\mathsf{SKE}_{\mathsf{detCPA}}$ is det-LoR-CPA-secure, then out_2 and out_3 are computationally indistinguishable.

As the proof is similar to the proof for claim 4, we omit it.

Claim 16 out_3 and out_4 are identically distributed.

For a proof, see the proof of the very similar claim 3.

Claim 17 If $\mathsf{SKE}_{\mathsf{detCPA}}$ is det-LoR-CPA-secure, then out_4 and out_5 are computationally indistinguishable.

As the proof is similar to the proof for claim 4, we omit it.

Claim 18 out_5 and out_6 are identically distributed.

Proof 16 As the changes between H_5 and H_6 are only syntactic and oblivious for the environment, the claim follows.

2.4. Implementation and Evaluation

We implemented the proposed private set intersection protocol with Intel SGX enclaves on an Intel Xeon E3-1275 v6 processor.[4]

Remote attestation of SGX enclaves is more involved than modeled in the ideal functionality. While the ideal functionality just signs enclave outputs with a globally known signature key-pair, SGX enclaves complement their output with data called attestation evidence. However, in contrast to the formal model, this data cannot be independently verified. SGX provides two implementations of attestations: EPID and DCAP. To check the correctness of an EPID attestation, the attestation evidence needs to be submitted to an Intel-operated service, called the Intel Attestation Service. That service decrypts the attestation evidence and then verifies the contained group signature. The result of that check is then reported back, together with an RSA-2048 signature with a fixed key. Such a web request takes approximately 300 ms. If a protocol needs many attestations by the same enclave, the necessity of many separate interactions with the attestation service can be sidestepped by initially generating a signature key pair, then

[4] For the code, see https://github.com/kastel-security/psi-with-sgx.

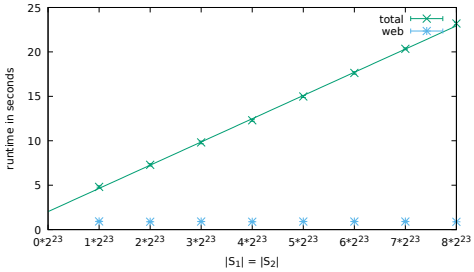

Figure 2.2.: Runtime of the PSI protocol, depending on the number of elements in the input sets for elements of 128 bit size. The size of S_1 and S_2 is equal and $|S_1 \cap S_2| = \frac{|S_1|}{2}$. The time is shown as total runtime including wait time for the Intel Attestation Service (roughly 0.8s) which is also shown for reference in the "web" series. Error bars are the standard derivation over 10 measurements.

performing remote attestation with the verification key as output, and then using this keypair to sign all of the output that needed attestation before. For the evaluation, we used EPID attestations but did not implement this optimization. ECDSA attestations realized with the Intel DCAP toolkit implicitly use this optimization, as only one initial attestation needs to be verified by the "Provisioning Certification Service" operated by Intel to attest an ECDSA signature keypair which is then used to sign and verify all further attestations. Additionally, cacheable attestation collateral needs to be downloaded from an Intel service to ensure that the attesting system is up-to-date.

For our benchmarks, we used EPID attestations, also because we found the responses from the Intel Attestations Service can be verified more easily from with enclaves than DCAP attestations. We use the RSA-2048 signature in the response from the Intel Attestation Service to check an attestation. Its required verification key is delivered into the enclaves by embedding it directly into the enclaves' code. Verifying DCAP attestations from within enclaves is more involved.

The runtime of the set intersection protocol consists of a constant part, mainly for doing the interactions with the Intel Attestation Service to confirm the validity of a remote enclave attestation and exchanging a key between the enclaves before the actual computation, and a part linear in the input size. fig. 2.2 shows the overall runtime for set intersection with 128-bit elements. The protocol for the different parties was executed in the same program with a simulated veth device introducing 10ms of latency for all TCP packets. To intersect 2^{24} 128-bit elements with another set of 2^{24} 128-bit elements, having half of the elements in the intersection, we get a total runtime of 7.3 seconds of which the linear regression suggests a constant overhead of 2.0 seconds. For comparison the maliciously secure protocol from Rindal et al. [92] takes around 8.0 seconds in their paper and takes around 10.7 seconds when run on our hardware.

The size of the enclave memory of SGX enclaves needs to be known ahead of time and additionally cannot be increased without limits as it becomes harder to keep this memory protected with a larger enclave state. Luckily, our protocol can easily be implemented with limited enclave state size by streaming the values for encryption/hashing through the enclave, not requiring the enclave memory size to depend on the sets' size. A modification is needed for P_2's enclave, which needs to keep a large set of values as internal state, and then choose the UHF and start outputting hashes. For the correctness, it is important that P_2 needs to be committed to all the values before the enclave chooses the UHF. To eliminate the large intermediate state, the enclave can just store a collision-resistant hash (we used SHA256) of the values. Later, when the UHF needs to be evaluated on that data, the data is supplied by P_2 again. After the hashes are calculated,

Protocol	Runtime	Network Model
Ours	7.3s	loopback + latency
Ours	8.1s	loopback + latency + 1Gbit/s
from PaXoS [90]	69s	loopback
from VOLE [92]	10.7s	in-process memory
from VOLE [92] + sort	11.2s	in-process memory

Figure 2.3.: Evaluation and comparisons executed on our hardware. The runtime was measured for inputs consisting of 2^{24} 128-bit elements on the same hardware.

the attestation on the universally hashed values is only released if the collision-resistant hash on the input is unchanged, compared to the one calculated in the commit phase. As passing data into enclaves is fast (enclaves can access non-enclave memory at will), this approach brings a significant performance improvement.

For an instantiation of the UHF, we use the 64-bit variant XXHash and for choosing a random hash function we choose the 64-bit seed of XXHash3 uniformly at random. PSI benchmarks usually assume constant-size elements of 128-bit. Deterministically encrypting 128-bit elements can be easily done with just a single AES operation. For sorting we use an optimized variant of radix sort. As x86-64 CPUs support 128-bit unsigned integers as native types, implementing a 128-bit radix sort is straightforward.

fig. 2.3 show the runtime of our PSI protocol compared to two recent maliciously secure PSI protocols [90, 92]. Further information on how the protocols were executed is given in section 2.4.1. Due to implementation differences and restrictions of our evaluation setup we did not execute all PSI protocols with the exact same network abstraction. The differences that remained are noted in the column "Network Model". In our implementation we use a real TCP socket connected over a loopback adapter that simulates latency (and optionally bandwidth limitation). For the implementation based on PaXoS we did not simulate latency, the implementation based on VOLE does not use sockets at all, but communicates via an in-memory buffer. The network model for our implementation is the most pessimistic one. For the last line in the table, we adjusted [92] to not leak the input set order anymore. Details can be found in section 2.4.2.

2.4.1. Evaluation Setup Details

This appendix contains details on programs we executed for evaluation, specifically for the numbers in fig. 2.3.

Our Implementation. For evaluating our code we, build the sgxsgk using the provided Docker file and with that the sgxdev container. After inserting the credentials for the SGX Developer account into GeneralSettings.h, we compile code and run the onesided-app binary. To setup the network devices with simulated latency, we use:

```
sudo ip link add pleft type veth peer name \
  pright
sudo ip link set pright up
sudo ip link set pleft up
sudo tc qdisc add dev pleft root netem \
```

```
delay 10ms
```

and then provide the ipv6 link-local address of one interface with the interface specifier of the other as only argument to the onesided-app binary.

PSI from PaXoS. We use the implementation provided by the authors https://github.com/cryptobiu/PaXoS_PSI with some adjustments and a Dockerfile applied by Phillipp Schoppmann on his fork of that repository: https://github.com/schoppmp/PaXoS_PSI. With a few minor adjustments on commit 57840266d, we are able to compile the implementation (see the provided diff).

Then we create a parties-file bin/Parties.txt:

```
cat bin/Parties.txt
party_0_ip = 127.0.0.1
party_1_ip = 127.0.0.1
party_0_port = 8000
party_1_port = 8200
```

Then we run bin/frontend partyID 1 internalIterationsNumber 10 hashSize 16777216 fieldSize 238 partiesFile bin/Parties.txt malicious 1 and bin/frontend partyID 1 internalIterationsNumber 10 hashSize 16777216 fieldSize 238 partiesFile bin/Parties.txt malicious 1.

PSI from VOLE. We use the source code from the authors in a clean Debian Bullseye container, install cmake build-essential python3 libtool and run the provided build.py. We run the provided exectable out/build/linux/frontend/frontend -perf -psi -mal -v 3 -fakeBase 0 -nn 24. To compare the runtime when sorting we add the radixsort.cpp from our repository and insert the sorting of hashes after the "time point" RsPsiSender::run-eval.

2.4.2. PSI Implementations Leaking the Order of the Input Set

The source code for [90] is provided on their GitHub repository[5]. The main orchestrating logic is implemented in ProtocolParty.cpp. Sender::computeXors calculates the hash values of the senders input (stored in the member variables keys and vals) and stores them one-by-one in xors (line 673). This list is then sent to the other party in line 690. This reveals the location of the elements which are in the intersection.

The source code for [92] is also provided on GitHub[6]. The main orchestration happens in RsPsi.cpp[7] and for the sender specifically in RsPsiSender::run, which receives the input set as parameter, evaluates the OKVS on it (line 66) to produce hashes in the same order and sends it to the receiver at the end of the method (line 90). For comparing a corrected version in the evaluation we added the same sorting code we used after the hashes were calculated to remove any order contained in them.

[5] https://github.com/cryptobiu/PaXoS_PSI
[6] https://github.com/Visa-Research/volepsi
[7] https://github.com/Visa-Research/volepsi/blob/63990d8b873622844bb0ac588ab19d2ca66f062e/volePSI/RsPsi.cpp

2.5. Further Applications

We present further applications of the model defined in section 2.2.

2.5.1. One-Sided Hamming Distance

Using a protocol that is somewhat reminiscent of our construction π_{oPSI} for one-sided PSI, parties P_1 and P_2 can compute the Hamming distance on bitstrings in a way that P_1 learns the distance, even if the TEEs are almost-transparent or passively corrupted (and P_1 and P_2 honest).

First of all, we need an additional cryptographic building block called message authentication codes (MACs), which can be seen as the symmetric analogue of digital signatures. As MACs are part of the attestation protocol of Intel SGX [37], we assume that MAC tags can be computed and verified securely and consider a variant of $\mathcal{G}_{att}$ that supports MACs as secure operations. In the following, we assume that MACs are length-normal, i.e. that for messages x_1, x_2 with $|x_1| = |x_2|$, the MAC tags of x_1 and x_2 have the same length.

Construction 2 (Protocol π_{HD}) Let Sig be a signature scheme, MAC a length-normal message authentication code, and $\mathsf{SKE}_{\mathsf{detCPA}}$ a deterministic secret-key encryption scheme. Let $\kappa \in \mathbb{N}$ be a security parameter. Let $\mathcal{G}_{att} = \mathcal{G}_{att}[\mathit{Sig}, \mathsf{SKE}_{\mathsf{detCPA}}, \mathit{MAC}, \kappa]$. In the following, we assume that plaintexts are appropriately padded such that all ciphertexts within a set have the same plaintext length.[8]

	Known to			
Key	P_1	P_2	TEE of P_1	TEE of P_2
k^1_{SKE}	-	✓	✓	✓
k^2_{SKE}	✓	-	✓	-
k^3_{SKE}	-	-	✓	-
k^4_{SKE}	✓	✓	-	-
k_{MAC}	✓	-	✓	-

Table 2.2.: Overview of the used keys.

Set	Contains	Encrypted with key(s)	Known to	Origin
C_1	S_1	k^4_{SKE}	P_1, TEE_1	-
M	$(0\|\|i)$ and $(1\|\|i)$	k^4_{SKE}	P_1, P_2, TEE_1	-
C'_1	S_1	$k^4_{\mathsf{SKE}}, k^3_{\mathsf{SKE}}, k^2_{\mathsf{SKE}}, k^1_{\mathsf{SKE}}$	P_1, P_2, TEE_1	C_1
M'	$(0\|\|i)$ and $(1\|\|i)$	$k^4_{\mathsf{SKE}}, k^3_{\mathsf{SKE}}, k^{\mathsf{SKE}}_1$	P_1, P_2, TEE_1	M
C''_1	S_1	$k^4_{\mathsf{SKE}}, k^3_{\mathsf{SKE}}, k^2_{\mathsf{SKE}}$	P_1, P_2	C'_1
M''	$(0\|\|i)$ and $(1\|\|i)$	$k^4_{\mathsf{SKE}}, k^3_{\mathsf{SKE}}$	P_2	M'
C_2	S_2	$k^4_{\mathsf{SKE}}, k^3_{\mathsf{SKE}}$	P_1, P_2	M''
C'''_1	S_1	$k^4_{\mathsf{SKE}}, k^3_{\mathsf{SKE}}$	P_1	C''_1

Table 2.3.: Overview of the used ciphertext sets.

[8] This is necessary to later reduce to the det-LoR-CPA property of $\mathsf{SKE}_{\mathsf{detCPA}}$.

Program prog_1 **of** P_1**'s enclave:**

1. Receive $(\mathrm{mpk}, \mathrm{eid}_1, \mathrm{eid}_2, \mathsf{prog}_2, \mathrm{vk})$ as first input.
2. Execute $(\texttt{key-exchange}, \mathrm{eid}_2)$. Eventually, receive a message $(\mathrm{eid}_1', \mathrm{eid}_2', \mathsf{hdl})$. Store hdl^1_{SKE} if $\mathrm{eid}_1' = \mathrm{eid}_1$ and $\mathrm{eid}_2' = \mathrm{eid}_2$.
3. On input $(\texttt{init}, \mathrm{sid}, C_1)$, store C_1.
4. On input $(\texttt{release-key}, \mathrm{sid})$:
   a) Execute $\mathsf{hdl}_{MAC} = \mathsf{mac}.\mathtt{gen}()$,
   b) $\mathsf{hdl}^2_{SKE} = \mathsf{ske}.\mathtt{gen}()$, $\mathsf{hdl}^3_{SKE} = \mathsf{ske}.\mathtt{gen}()$,
   c) $k_{MAC} = \texttt{release-key}(\mathsf{hdl}_{MAC})$, $k^2_{SKE} = \texttt{release-key}(\mathsf{hdl}^2_{SKE})$, and output (k_{MAC}, k^2_{SKE}).
5. On input $(\texttt{msg}, \mathrm{sid}, M, \sigma_M)$:
   a) Verify the signature σ_M using the key vk, which part of prog_1. If the signature verification fails, halt.
   b) Check that M and C_1 are consistent and valid, i.e.
      i. The tuples in M are unique, have a correct MAC under k_{MAC} and
      ii. for every ciphertext in C_1, there exists a tuple in M with the corresponding ciphertext.
      iii. If this is not the case, halt.
   c) For $i = 1, \ldots, |C_1|$, set
      i. $c_i' = \mathsf{ske}.\mathtt{enc}(\mathsf{hdl}^3_{SKE}, c_i)$, $\tau_i' = \mathsf{mac}.\mathtt{mac}(\mathsf{hdl}_{MAC}, c_i')$,
      ii. $c_i'' = \mathsf{ske}.\mathtt{enc}(\mathsf{hdl}^2_{SKE}, (c_i', \tau_i'))$,
      iii. $c_i''' = \mathsf{ske}.\mathtt{enc}(\mathsf{hdl}^1_{SKE}, c_i'')$ and
      iv. $C_1' = (c_1''', \ldots, c'''_{|C_1|})$.
   d) For $j = 1, \ldots, |M|$, set
      i. $c_j^* = \mathsf{ske}.\mathtt{enc}(\mathsf{hdl}^3_{SKE}, c_m^j)$, $\tau_j^* = \mathsf{mac}.\mathtt{mac}(\mathsf{hdl}_{MAC}, c_j^*)$, where c_m^j is the j-th ciphertext contained in M,
      ii. $c_j^{**} = \mathsf{ske}.\mathtt{enc}(\mathsf{hdl}^1_{SKE}, (c_j^*, \tau_j^*))$ and
      iii. $M' = (c_1^{**}, \ldots, c^{**}_{|M|})$
   e) Output (C_1', M').

   All steps are performed using the built-in cryptographic operations such that no intermediate results are visible.
6. Ignore all further messages.

Program prog_2 **of** P_2**'s enclave:**

1. Receive $(\text{mpk}, \text{eid}_1, \text{eid}_2, \text{prog}_2)$ as first input.
2. Eventually receive a message $(\text{eid}_1', \text{eid}_2', \mathsf{hdl})$. Store hdl if $\text{eid}_1' = \text{eid}_1$ and $\text{eid}_2' = \text{eid}_2$.
3. On input $(\texttt{release-key}, \text{sid})$, return $k_{SKE}^1 = \texttt{release-key}(\mathsf{hdl})$.
4. Ignore all further messages.

Protocol:

If activated without input, a party yields the execution until input is received.

1. Each party P_i (eventually) receives input $(\texttt{input}, \text{sid}, S_i)$ and is parameterized with a security parameter κ.
2. P_1 and P_2 each send $(\texttt{getpk})$ to $\mathcal{G}_{att}$ to obtain the master public key mpk.
3. P_1 and P_2 call $\mathcal{F}_{\text{KE}}$ once, using SID sid||1 (with P_1 acting as initiator) to obtain a key k_{SKE}^4.
4. P_2 generates a signature key pair $(\text{sk}, \text{vk}) \leftarrow \mathit{Sig.Gen}(1^\kappa)$ and sends vk to P_1.
5. P_1 sends $(\texttt{install}, \text{sid}, \text{prog}_1)$ to $\mathcal{G}_{att}$, where prog_1 is the program for P_1's enclave in construction 1, parameterized with the verification key vk received from P_2. It obtains eid_1 as answer and sends eid_1 to P_2.
6. Similarly, P_2 installs an enclave with program prog_2, obtains eid_2 and sends it to P_1.
7. P_1 sends $(\text{mpk}, \text{eid}_1, \text{eid}_2, \text{prog}_2)$ to its TEE with EID eid_1 via $\mathcal{G}_{att}$. Conversely, P_2 sends $(\text{mpk}, \text{eid}_1, \text{eid}_2, \text{prog}_1)$ to its TEE with EID eid_2.
8. Let $S_1 = b_1^1 || \ldots || b_1^{n_1}$ with $b_1^i \in \{0,1\}$. For $i = 1, \ldots, n_1$, let $c_1^i = \mathsf{SKE}_{\text{detCPA}}.\mathit{Enc}(k_{SKE}^4, b_1^i || i)$. P_1 sends $(\texttt{init}, \text{sid}, C_1 = \{c_1^1, \ldots c_1^{n_1}\})$ to the TEE with EID eid_1.
9. P_1 sends $(\texttt{release-key}, \text{sid})$ to its TEE with EID eid_1, obtaining keys k_{MAC} and k_{SKE}^2.
10. For $i = 1, \ldots, n_1$, $j \in \{0,1\}$, P_1 computes $c_1^{i,j} = \mathsf{SKE}_{\text{detCPA}}.\mathit{Enc}(k_{SKE}^4, j||i)$ and $\tau_1^{i,j} \leftarrow \mathit{MAC.MAC}(k_{MAC}, c_1^{i,j})$, sets $M = \{(c_1^{1,0}, \tau_1^{1,0}), \ldots, (c_1^{n_1,1}, \tau_1^{n_1,1})\}$, sorts M lexicographically and sends the sorted M to P_2.
11. On receiving M, P_2 asserts that for $i = 1, \ldots, |M/2|$ and $j = 0, 1$, there is an element $(c_1^{i,j}, \tau_1^{i,j})$ in M such that $c_1^{i,j} = \mathsf{SKE}_{\text{detCPA}}.\mathit{Enc}(k_{SKE}^4, j||i)$. If this is the case, P_2 computes $\sigma_M \leftarrow \mathit{Sig.Sign}(\text{sk}, M)$ and sends σ_M to P_1.
12. P_1 sends $(\texttt{msg}, \text{sid}, M, \sigma_M)$ to its TEE with EID eid_1. It then receives $(\texttt{output}, \text{sid}, (C_1', M'), \sigma_2)$ and sends $((C_1', M'), \sigma_2)$ to P_2.
13. On receiving $((C_1', M'), \sigma_2)$, P_2 first verifies the signature σ_2, verifying that prog_1 is parameterized with vk. Upon success, it sends $(\texttt{release-key}, \text{sid})$ to its TEE with EID eid_2 and obtains a key k_{SKE}^1 and uses k_{SKE}^1 to decrypt the elements in C_1' and M', resulting in sets C_1'' and M''. Then, P_2
    a) permutes M'' such that M'' is ordered by $(j||i)$ again (which is possible because P_2 can reconstruct the applied permutation in item 11) and
    b) sets C_2 to be the set of elements in M'' that belong to its input, i.e. $(b_2^i||i)$,
    c) sorts C_1'' and C_2 lexicographically and then

d) sends the sorted C_1'' and C_2 to P_1.

14. On receiving C_1'' and C_2, P_1 decrypts C_1'' to C_1''' using k^2_{SKE} and then checks that i) all elements in C_2 resp. C_1''' are unique and ii) all MAC tags in C_2 and C_1''' are valid, using key k_{MAC}. Let d denote the number of elements that are both contained in C_2 and C_1'''. Finally, P_1 outputs d, which is the Hamming distance of the inputs of P_1 and P_2.

If P_1 is corrupted, the simulator can extract P_1's input by reading the set of ciphertext C_1 sent to the TEE and decrypting it using k^4_{SKE}. Then, it learns the Hamming distance d from the ideal functionality. The set C_2 is simulated by decrypting C_1'' using k^2_{SKE} and reusing d elements. The other elements of C_2 are used from M''.

If P_2 is corrupted, the simulator can extract P_2's input from M'' and C_2. The inputs of P_1 to its TEE can be emulated by sending a dummy string, as P_2 does not learn the result.

If the TEEs are corrupted semi-honestly, it suffices to execute the protocol honestly on dummy inputs for P_1 and P_2 that have the appropriate length as reported by the ideal functionality.

2.5.2. Trusted Initializer

A popular application of TEEs is the use as a trusted initializer [76], which provides protocol parties with correlated randomness.

In this setting, leakage to the manufacturer is usually unproblematic, as the correlated randomness is independent of the parties' inputs and the TEEs are not used for further computation.

However, side-channels may pose a problem, as they could allow one party to learn "too much". To this end, consider the following protocol, somewhat similar to the construction in [76]. Let PRG be a pseudorandom generator and let Sig be a signature scheme.

1. Parties P_1 and P_2 are each equipped with a TEE.
2. The TEEs initially exchange keys k_0, k_1.
3. The TEE of P_1 computes $m_b = \mathsf{PRG}(k_b)$ for $b \in \{0,1\}$ and outputs (m_0, m_1) to P_1.
4. The TEE of P_2 samples a random $b' \xleftarrow{\$} \{0,1\}$ and outputs $(b', m_{b'} = \mathsf{PRG}(k_{b'}))$ to P_2.

This randomized OT can later be de-randomized. If the TEE of P_2 has side-channels that expose k_0 and k_1, the protocol is completely insecure, as P_2 would learn both m_0 and m_1.

We propose a construction for randomized oblivious transfer which is not only secure for semi-honest TEEs, but also for almost-transparent TEEs.

Construction 3 (Protocol π_{rOT}) Let Sig be a signature scheme and $\mathsf{SKE}_{\mathsf{detCPA}}$ a deterministic secret-key encryption scheme. Let $\kappa \in \mathbb{N}$ be a security parameter. Let $M = \{0,1\}^\kappa$ be a message space. Let $\mathcal{G}_{att} = \mathcal{G}_{att}[\mathit{Sig}, \mathsf{SKE}_{\mathsf{detCPA}}, \kappa]$.

Program prog_1 **of** P_1**'s enclave:**

1. Receive $(\text{mpk}, \text{eid}_1, \text{eid}_2, \text{prog}_2)$ as first input.
2. Execute $(\texttt{key-exchange}, \text{eid}_2)$. Eventually, receive a message $(\text{eid}_1', \text{eid}_2', \mathsf{hdl})$. Store hdl if $\text{eid}_1' = \text{eid}_1$ and $\text{eid}_2' = \text{eid}_2$.
3. On input $(\texttt{init}, \text{sid}, \sigma, \text{vk})$:
   a) Abort if $\mathit{Sig.Vfy}(\texttt{ready}, \sigma, \text{vk}) \neq 1$.
   b) Sample $m_0, m_1 \xleftarrow{\$} M$.
   c) Set $c_i = \texttt{ske.enc}(\mathsf{hdl}, m_i)$ for $i \in \{0, 1\}$.
   d) Output $(\texttt{ciphertexts}, \text{sid}, (c_0, c_1, \text{vk}))$.
   e) Upon the next activation, output $(\texttt{result}, \text{sid}, m_0, m_1)$.
4. Ignore all further messages.

Program prog_2 **of** P_2**'s enclave:**

1. Receive $(\text{mpk}, \text{eid}_1, \text{eid}_2, \text{prog}_2)$ as first input.
2. Eventually receive a message $(\text{eid}_1', \text{eid}_2', \mathsf{hdl})$. Store hdl if $\text{eid}_1' = \text{eid}_1$ and $\text{eid}_2' = \text{eid}_2$.
3. On input $(\texttt{init}, \text{sid}, (\texttt{ciphertexts}, \text{sid}, (c_0, c_1), \text{vk}), \sigma')$:
   a) Verify that σ' is a valid signature for the output $(\texttt{ciphertexts}, \text{sid}, (c_0, c_1), \text{vk})$ of the TEE with EID eid_1. If the verification fails, halt.
   b) Sample $b \xleftarrow{\$} \{0, 1\}$.
   c) Set $m_b = \texttt{ske.dec}(\mathsf{hdl}, c_b)$.
   d) Output $(\texttt{result}, \text{sid}, (b, m_b))$.
4. Ignore all further messages.

Protocol:

If activated without input, a party yields the execution until input is received.

1. Each party P_i (eventually) receives input $(\texttt{start}, \text{sid})$.
2. P_1 and P_2 each send $(\texttt{getpk})$ to $\mathcal{G}_{att}$ to obtain the master public key mpk.
3. P_2 generates a signature key pair $(\text{sk}_2, \text{vk}_2) \leftarrow \mathit{Sig.Gen}(1^\kappa)$ and sends vk_2 to P_1.
4. P_1 sends $(\texttt{install}, \text{sid}, \text{prog}_1)$ to $\mathcal{G}_{att}$, where prog_1 is the program for P_1's enclave in construction 3. It obtains eid_1 as answer and sends eid_1 to P_2.
5. Similary, P_2 installs an enclave with program prog_2, obtains eid_2 and sends it to P_1.
6. P_1 sends $(\text{mpk}, \text{eid}_1, \text{eid}_2, \text{prog}_2)$ to its TEE with EID eid_1 via $\mathcal{G}_{att}$. Conversely, P_2 sends $(\text{mpk}, \text{eid}_1, \text{eid}_2, \text{prog}_1)$ to its TEE with EID eid_2.
7. P_1 samples $r_0, r_1 \xleftarrow{\$} M$ and sends (r_0, r_1) to P_2 via $\mathcal{F}_{\text{SMT}}$, i.e. via a secure channel.

8. Upon receiving (r_0, r_1) via $\mathcal{F}_{\text{SMT}}$, P_2 samples a random bit $b' \xleftarrow{\$} \{0,1\}$ and sends b' to P_1 via $\mathcal{F}_{\text{SMT}}$.
9. P_2 also generates a signature to the message "ready", i.e. computes $\sigma_2 \leftarrow \mathit{Sig.Sign}(\text{sk}_2, \texttt{ready})$. Then, P_2 sends σ_2 to P_1.
10. Upon receiving σ_2, P_1 sends $(\texttt{init}, \text{sid}, \sigma_2, \text{vk}_2)$ to its TEE with EID eid_1, receives $(\texttt{ciphertexts}, \text{sid}, (c_0, c_1, \text{vk}_2), \sigma_C)$ as answer and forwards it to P_2.
11. On its next activation P_1 activates its TEE with EID eid_1 again and receives $(\texttt{result}, \text{sid}, m_0, m_1)$. Then, P_1 outputs $(\texttt{result}, \text{sid}, (m_0 \oplus r_0, m_1 \oplus r_1)$ if $b' = 0$ and $(\texttt{result}, \text{sid}, (m_1 \oplus r_1, m_0 \oplus r_0)$ otherwise, i.e. if $b' = 1$.
12. Upon receiving $(\texttt{ciphertexts}, \text{sid}, (c_0, c_1, vk'), \sigma_C)$ from P_1, P_2 asserts that $\text{vk}' = \text{vk}_2$ and verifies σ_C. In case of failure, abort. Then, P_2 sends $(\texttt{init}, \text{sid}, (\texttt{ciphertexts}, \text{sid}, (c_0, c_1, vk'), \sigma_C))$ to its TEE with EID eid_2 and receives $(\texttt{result}, (b, m_b))$ as answer. P_2 then outputs $(\texttt{result}, b \oplus b', m'_b \oplus r_b)$.

We consider the following functionality, where M is a message space.

- If P_1 is honest, choose $m_0, m_1 \xleftarrow{\$} M$ uniformly at random. If P_1 is corrupted, the adversary may provide $m_0, m_1 \in M$.
- If P_2 is honest, choose $b \xleftarrow{\$} \{0,1\}$ uniformly at random. If P_2 is corrupted, the adversary may provide $(b, m_b) \in \{0,1\} \times M$.
- Generate a private delayed output (m_0, m_1) to P_1 and a private delayed output (b, m_b) to P_2.

This functionality provides meaningful security for P_1 because P_2 never learns m_{1-b} and for P_2 because P_1 never learns b.

If P_1 is corrupted, the simulator simulates P_2 honestly and observes the protocol output for P_1. It sends the output to the ideal functionality as m_0, m_1.

Similarly, if P_2 is corrupted, the simulator simulates P_1 honestly and observes the output for P_2. Then, it sends this output to the functionality as (b, m_b).

If the TEE is corrupted semi-honestly, the simulator simply executes the protocol honestly on behalf of P_1 and P_2 and reports all messages from the TEE functionality.

The key insight with respect to the security in the presence of almost-transparent TEEs is the fact the TEE of P_2 selectively decrypts only one of the ciphertext, with the other ciphertext's plaintext remaining hidden. In a sense, this is the computational analogue of an erasure channel.

2.5.3. Oblivious Transfer

construction 3 can be easily modified to directly compute oblivious transfer for user-provided inputs (m_0, m_1) of P_1 and b of P_2.

2.6. Conclusion

Private set intersection is an important privacy-enhancing technology with many possible applications. In this work, we have constructed and implemented a protocol for one-sided two-party PSI using trusted execution environments that is both asymptotically and practically efficient, beating the best known protocol with respect to single-threaded performance.

Instead of fully trusting the TEE, we substantially lower the required trust: Our protocol remains secure even if either i) the TEEs have side-channels and leak information to the host, except for simple cryptographic operations that remain secure or ii) the TEEs are semi-honest and leak all information to an entity such as the manufacturer, but not the protocol participants. This is motivated by what is assumed to be provided by current TEEs such as Intel SGX.

To show the usefulness of our model, we have also presented protocols in our model for additional tasks such as computing the one-sided Hamming distance or (randomized) oblivious transfer.

3. Decision Tree Training

This chapter is based on joint work with Robin Berger and Alexander Koch published at ACNS 2024[13]. In this work I have made significant contribution to the core protocol and the security proof. The analysis of how security of this approach relates to different trust assumptions of the hardware, is also my contribution.

3.1. Introduction

Privacy-preserving machine learning has gained a lot of traction in recent years, due to the tremendous benefits of having automated data-driven decision making, while caring for the legitimate privacy interests of the user, especially in a multiparty setting. More importantly, due to legal requirements, many uses of machine learning (ML) algorithms would not even be possible in a setting where access to sensitive information (such as medical data) is required.

One ML algorithm that is relatively popular, due to its simplicity and interpretability, is decision tree learning. It is usually employed in the more general decision forest version. Here, we use a training data set of entries with many attributes and a label, to build a decision tree. This tree branches based on thresholds w.r.t. the attribute values, and has labels annotated to its leaf nodes. To classify a new entry, one follows the tree from the root to a leaf node by comparing the attributes against the thresholds. The classification result is the label annotated to the reached leaf node.

Training such trees via secure multiparty computation has already been proposed by [2, 53]. Because this is in a strong privacy model, these protocols are relatively expensive regarding computation and round complexity. The overall round complexity of [53] is $O(h(\log m + \log n))$, where h is the a priori fixed depth of the tree, m is the number of attributes and n is the number of data entries. Hence, in the setting where they add a realistic 50 ms delay to each message on the network, the overall running time of the protocols increases by several orders of magnitude.

This motivates the following research question: Can we greatly improve the performance and round complexity of the protocol by allowing for some leakage. (This leakage can later be partly avoided again, with the help of secure enclaves.) Ideally, we would want a small constant number of rounds.

As comparisons or sorting are the main ingredients to Decision Tree training algorithms, we propose to use an extended version of Order-Revealing Encryption (ORE). In ORE schemes, the values are encrypted using a secret key, but given two ciphertexts, one can evaluate the order of the messages they encrypt without knowing this key. We extend this by a way to update the ciphertexts to a new key, in a setting where the key space is a multiplicative group. Hence, given a ciphertext $c = \mathsf{Enc}(k, m)$ and second key k', one can run $\mathsf{Upd}(k', c)$ to obtain a new ciphertext c' that is equal to $\mathsf{Enc}(k \cdot k', m)$, making use of key-homomorphic pseudorandom functions (PRFs).

Using this new primitive, which we believe to be of independent interest, we construct a conceptually simple protocol, using only three rounds, that allows two parties to jointly compute a decision tree on their data. Here, the two parties A and B both have a horizontal partition of the data set and B does the main tree computation. In a nutshell this works as follows:

- To have all data points ORE-encrypted under the same keys (using one key per attribute), the parties proceed in a Diffie-Helman key exchange-like manner: B encrypts his data under his keys k_j. A updates these ciphertexts with her keys k'_j, obtaining ciphertexts under $k_j \cdot k'_j$. B then updates these ciphertexts using k_j^{-1}, obtaining ciphertexts under keys k'_j.
- A also sends her own data points encrypted under k'_j to B, together with the used labels (outcome attributes).
- Now, B has all the data points under the same keys k'_j, and can use the comparison function and the labels to compute a decision tree that uses the encrypted values as thresholds.
- B sends this encrypted tree to A, who can now decrypt it using her keys k'_j.

This is a three-round protocol in the honest-but-curious setting, and hence much faster than the MPC protocols of Hamada et al. [53] and Abspoel, Escudero, and Volgushev [2], if one assumes plausible latency. However, this also comes at the cost of considerable overall leakage due to what can be inferred from the comparisons (and the leakage of the ORE scheme), if the scheme is used in the bare (non-enclave) version. We give a more general analysis on the leakage, as well as an information-theoretic upper bound thereof in section 3.4.

3.1.1. Related Work

Order-Revealing Encryption Order-Revealing Encryption (ORE) was introduced by Boneh et al. [16] as a more flexible and more secure notion of Order-Preserving Encryption (OPE). In contrast to OPE, where the natural ordering of ciphertexts must be identical to the natural ordering of the messages they encrypt, Order-Revealing Encryption allows to define a dedicated comparison function on the ciphertexts for evaluating the natural order of the elements contained within the ciphertexts. The main motivation for this was to enable efficient search operations in encrypted databases. Following the introduction of ORE, Chenette et al. [35] formalized security of ORE schemes, as well as giving a construction of such a scheme. This construction inspired a new scheme by Lewi and Wu [70], which is in a slightly different setting, namely the left–right framework, where there are "left" ciphertexts and "right" ciphertexts and a left ciphertext can be compared only with a right ciphertext. To allow for ORE schemes to be used in a multi-user setting, Li, Wang, and Zhao [71] introduced the notion of delegatable ORE schemes, where it is possible to issue comparison tokens, which allows one to compare ciphertexts of different users. In a similar way, Lv et al. [77] extend ORE schemes to a multi-user setting. One problem with this approach, however, is that if one party has a comparison token allowing another party's ciphertexts to be compared to her own, it can decrypt the other party's ciphertexts again.

As ORE schemes allow comparisons of elements, they inherently leak information about the messages encrypted in ciphertexts, as soon as more than one message is encrypted under the same key. This has led to several investigations on how severely this leakage affects the data privacy. Grubbs et al. [51] and Durak, DuBuisson, and Cash [45] have shown that under some circumstances, ORE schemes provide no meaningful security. Jurado, Palamidessi, and Smith [62] however show that ORE schemes still provide some security if the message space is significantly larger than the amount of messages encrypted.

Privacy-Preserving Machine Learning Since machine learning models have become more widespread, there has been work towards being able to perform the training process thereof in a privacy-preserving manner. Several machine learning models have been considered in this setting, ranging from simple regression tasks [33, 63] to neural networks. Approaches for the latter include Fully-Homomorphic Encryption (FHE) [69], and MPC protocols [64]. Multiparty computation has also been used for training decision trees, where it is the most prevalent approach since the work by Lindell and Pinkas [73]. In their work, they discuss how MPC protocols can use the ID3 algorithm for training a decision tree. Since then, there have been several improvements over this work [43, 57].

Since the ID3 algorithm only supports discrete attributes, these approaches are not applicable to a setting with with continuous attributes. To overcome this issue, [2] use a variation of the C4.5 algorithm, which also supports continuous attributes. Their protocol works by computing the training process for each possible node in the resulting decision tree. However, this results in the runtime of their protocol being linear in the maximum number of possible nodes in a decision tree, and therefore exponential in the depth of the tree. Hamada et al. [53] improve over this with a protocol, which is linear in the depth of the tree, by partitioning the dataset in an oblivious way and performing the training once for each layer, considering this partitioning. This comes at the cost of requiring many network rounds. While these two state-of-the-art-approaches perform well under ideal circumstances, due to their runtime [2] is not applicable in a setting, where a high-depth decision tree is to be trained and [53] is not applicable in a high-latency environment.

Privacy-Preserving protocols based on fully-homomorphic encryption (FHE) are mostly restricted to decision tree evaluation (e.g. [36, 46]) and do not consider the secure training of decision trees. Most of those papers considering training a decision tree, consider a different setting. For example [5] consider the setting, where their goal is to outsource the training to a server, while some of the computationally intensive steps (for FHE) are still done by the client. Hence, it is not directly comparable to our work. Vaidya et al. [103] consider a setting, where the training data is partitioned vertically and FHE is only used to evaluate a heuristic in the training process, but the remaining computation is done in plain text. So far, there are no efficient decision tree training protocols, that perform the entire training using FHE. This is due to the fact that comparisons are computationally expensive in FHE. Indeed, Liu et al. [75] aim to provide a FHE-bsed protocol allowing comparisons in a multi-user setting, however a single plaintext to ciphertext comparison in their case takes a computation time of 100 ms and 1 s for a ciphertext-ciphertext comparison, rendering this approach infeasable in our setting.

3.1.2. Our Contribution

Our main contributions are as follows:

- We extend the notion of Order-Revealing Encryption (ORE) to Updatable ORE and give an instantiation thereof.
- Using this Updatable ORE scheme, we construct a three-round two-party protocol to compute a decision tree on a horizontal partitioning of a dataset with both parties providing training data. With this approach, we can apply the same training algorithms as used in plaintext training.

- We describe how this protocol can be combined with enclaves providing different security guarantees, in order to eliminate or reduce the introduced ORE leakage and to make the protocol actively secure. We also use an information-theoretic approach to quantify and give an upper bound for the ORE leakage.
- We implemented and experimentally verified the efficiency of our protocol, showing that it is faster than current state-of-the-art protocols, while achieving this speedup at the cost of some information leakage.

3.1.3. Outline

We introduce necessary preliminaries including the universal composability (UC) model, ORE and decision tree learning in section 1.3. We propose our notion of Updatable Order-Revealing encryption and also present a construction and a security proof in section 3.2. Section 3.3 contains the decision tree learning protocol, together with its security proof, and a remark on how to translate it into an actively-secure version using secure enclaves in a graceful-degradation manner. In section 3.4, we discuss the implications of the ORE leakage on the protocol. Finally in section 3.5, we evaluate our constructions based on a practical implementation.

3.2. Updatable Order-Revealing Encryption

We now augment the definition of ORE to allow for updating a ciphertext from one key to another, while retaining the messages contained in the ciphertexts.

Definition 14 (Updatable ORE) A 4-tuple of PPT algorithms $\mathsf{ORE} = (\mathit{Gen}, \mathit{Enc}, \mathit{Cmp}, \mathsf{Upd})$ is an Updatable ORE (uORE) scheme over key space $\mathcal{K} = \mathbb{Z}_p^\times$, message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$, if

- $(\mathit{Gen}, \mathit{Enc}, \mathit{Cmp})$ is an ORE scheme over key space $\mathcal{K}$, message space $\mathcal{M}$, and ciphertext space $\mathcal{C}$.
- $\mathsf{Upd}(k, c)$ takes a key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$ as input and outputs a new ciphertext $c' \in \mathcal{C}$.
- Enc and Upd are deterministic.

ORE is correct, if $(\mathit{Gen}, \mathit{Enc}, \mathit{Cmp})$ is a correct ORE scheme and satisfies the updatability property:

$$\forall k, k' \in \mathcal{K} \colon \mathsf{Upd}(k', \mathit{Enc}(k, m)) = \mathit{Enc}(k \cdot k', m)$$

Moreover, ORE is a secure uORE scheme w.r.t. a leakage function $\mathcal{L}$ iff $(\mathit{Gen}, \mathit{Enc}, \mathit{Cmp})$ is a secure ORE scheme w.r.t. $\mathcal{L}$.

Note that in our definition, we require that the key space $\mathcal{K} \coloneqq \mathbb{Z}_p^\times$. This means that any key k is invertible modulo p.

For our construction of an uORE scheme, we adapt the scheme from [35]:

Construction 4 Let $(\mathbb{G}, p, [1])$ be an additive group with prime order $p > 3$ and let $\mathsf{PRF}\colon \mathbb{Z}_p \times \{0,1\}^* \to \mathbb{G}$ be a key-homomorphic PRF with key space $\mathbb{Z}_p$ and message space $\{0,1\}^*$. Then, we define the uORE scheme $\mathsf{ORE} = (\mathit{Gen}, \mathit{Enc}, \mathit{Cmp}, \mathsf{Upd})$ with message space $\mathcal{M} = \{0,1\}^n$ for a parameter n, and ciphertext space $C = \mathbb{G}^n$, as follows:

- $\mathit{Gen}(1^\kappa)$: Return a uniformly random $k \leftarrow \mathbb{Z}_p^\times$
- $\mathit{Enc}(k, m = (m_1, \ldots, m_n))$: For $i = 1, \ldots, n$, set

  $$u_i = (1 + m_i) \cdot \mathsf{PRF}(k, (m_1, \ldots, m_{i-1})).$$

  Return $ct = (u_1, \ldots, u_n)$.
- $\mathit{Cmp}(ct = (u_1, \ldots, u_n), ct' = (u'_1, \ldots u'_n))$: Find the smallest i, such that $u_i \neq u'_i$. If such an i exists and $u'_i = 2 \cdot u_i$, return 1. Otherwise, return 0.
- $\mathsf{Upd}(k', ct = (u_1, \ldots, u_n))$: Set $u'_i = k' \cdot u_i$. Return $ct' = (u'_1, \ldots, u'_n)$.

Our construction is similar to the one by Chenette et al. [35]. In both cases, the key generation algorithm $\mathsf{Gen}(1^\kappa)$ samples a random element from the PRF key space, and $\mathsf{Enc}(k, (m_1, \ldots, m_n))$ is done bit by bit, by first computing $u'_i = \mathsf{PRF}(k, (m_1, \ldots, m_{i-1}))$ and then returning $u_i = u'_i$ if $m_i = 0$. If $m_i = 1$, both schemes return $u_i = \pi(u'_i)$ for an efficiently invertible permutation π. In both schemes, $\mathsf{Cmp}((u_1, \ldots, u_n), (u'_1, \ldots, u'_n))$ works by identifying the smallest i, for which $u_i \neq u'_i$ and checking if $u'_i = \pi(u_i)$ with the same permutation.

Comparing these two schemes, the only two differences are the PRF and the permutation π being used. In our scheme, we require the PRF to be key-homomorphic, which does not need to be the case in their scheme. This allows them to use $\mathbb{Z}_3$ as output space of the PRF and $\mathbb{Z}_3^n$ as the ciphertext space, whereas our used ciphertext space is $\mathbb{G}^n$. Moreover, we use $\pi(x) = 2 \cdot x$ as a permutation, whereas in their scheme, $\pi(x) = x + 1 \mod 3$ is used.

Because of this similarity, their security proof and ORE simulator also applies to our construction, when adjusting the permutation. Hence, both schemes are secure under the same leakage function:

Theorem 2 construction 4 is secure with the leakage function

$$\mathcal{L}(m_1, \ldots, m_t) = \{(i, j, \mathsf{hsb}(m_i \oplus m_j)) \mid m_i < m_j\},$$

where $\mathsf{hsb}(x)$ returns the position of the highest set bit of x.

Because of the similarity to the proof of [35], we will only give a proof intuition: For (u)ORE security to hold, there needs to be a simulator $\mathcal{S}^{\mathsf{ORE}}$ that, given only the leakage of messages, needs to be able to generate ciphertexts that are indistinguishable from encryptions of the messages with a random but (during the games) fixed key. In a first step, one replaces the PRF with a random function via lazy sampling. When asked to generate the first ciphertext, $\mathcal{S}^{\mathsf{ORE}}$ samples n elements from the output space of the PRF uniformly at random and outputs them as the first ciphertext. When asked to generate ciphertexts for any subsequent messages, it learns the position of the leftmost differing bit between this message and previous messages, as well as the message bit at these positions. This allows it to answer with consistent parts of ciphertexts, where message prefixes are equal, and with $\pi(x)$ or $\pi^{-1}(x)$ where the most significant difference is. Finally, for all other positions, for which no common prefix with another message exists, it

proceeds as in the case for the first message, sampling and outputting elements from the output space of the PRF.

Since this proof only requires the security of PRF and the efficient computation/inversion of π, the proof from [35] directly translates to our setting.

Remark 10 The leakage $\mathcal{L}$ of our scheme is actually sufficient to use faster sorting algorithms than plain comparison-based algorithms. For example MSD radix sort uses exactly the information provided by $\mathcal{L}$ to allow sorting in linear time. Sorting all ciphertexts by the first bit is easy, as there are only two comparable group elements for the first bit. After that, sorting the two partitions after the second bit is possible with the same approach. This reduces the amount of group operations required for sorting from $O(n \log n)$ to $O(n)$. This is especially interesting, as a main use-case for ORE are encrypted databases, where sorting is a major concern. Large databases often use advanced sorting algorithms that are not comparison-based, so the additional leakage of $\mathcal{L}$ over $\mathcal{L}_{\text{ideal}}$ can be used to speed up sorting.

Theorem 3 construction 4 is a correct ORE scheme.

Proof 17 Fix two messages $m, m' \in \{0,1\}^n$ with $m = (m_1, \ldots, m_n)$ and $m' = (m'_1, \ldots, m'_n)$. Then, we show that the correctness property holds for any $k \leftarrow \mathit{Gen}(1^\kappa)$, $(u_1, \ldots, u_n) \leftarrow \mathit{Enc}(k, m)$ and $(u'_1, \ldots, u'_n) \leftarrow \mathit{Enc}(k, m')$. We consider each case separately.

$m < m'$: In this case, there exists an i, such that $m_j = m'_j$ for $j < i$ and $m_j = 0$ and $m'_j = 1$. In this case, it holds that $u_j = u'_j$ for $j < i$. For PRF output $o = \mathsf{PRF}(k, (m_1, \ldots, m_{i-1}))$, and by definition of Enc, it holds that $u_i = o$ and $u'_i = 2 \cdot o$. If $o \neq 0_{\mathbb{G}}$, u_i and u'_i are different and Cmp returns 1 in this case. The probability for the event that $o = 0_{\mathbb{G}}$ is negligible (which follows from the fact that this probability is $1/p$ for a random function and because PRF is a PRF). Therefore, Cmp will output 1 with overwhelming probability.

$m = m'$: In this case $u_i = u'_i$ for all i, as Enc is deterministic and Cmp will output 0.

$m > m'$: Similarly to the first case, there exists an i, such that $m_j = m'_j$ for $j < i$ and $m_j = 1$ and $m'_j = 0$. With the same argument as in the case for $m < m'$, we know that $u_j = u'_j$ for $j < i$ and $u_i = 2 \cdot u'_i$. Since $\mathbb{G}$ is of prime order with $p > 3$, it also holds that $u'_i \neq 2 \cdot u_i = 2 \cdot 2 \cdot u'_i$ and therefore Cmp returns 0 with overwhelming probability.

Theorem 4 construction 4 satisfies the updatability property.

Proof 18 The updatability follows from the key-homomorphism of PRF. For all $k, k' \in \mathcal{K}$ and $m \in \{0,1\}^n$, it holds that

$$\begin{aligned}
\mathsf{Upd}(k', \mathit{Enc}(k, m)) &= \mathsf{Upd}(k', ((1 + m_i) \cdot \mathsf{PRF}(k, m_{1..i-1}))_{i=1,\ldots,n}) \\
&= (k' \cdot (1 + m_i) \cdot \mathsf{PRF}(k, m_{1..i-1}))_{i=1,\ldots,n} \\
&= ((1 + m_i) \cdot \mathsf{PRF}(k' \cdot k, m_{1..i-1}))_{i=1,\ldots,n} \\
&= \mathit{Enc}(k' \cdot k, m),
\end{aligned}$$

where $m_{1..i-1} \coloneqq (m_1, \ldots, m_{i-1})$.

- Upon receiving input $(\texttt{Input}, n_{\mathsf{B}}, m^{\mathsf{B}}_{1,1}, \ldots m^{\mathsf{B}}_{n_{\mathsf{B}},X})$ from B, send $(\texttt{InputReceived}, n_{\mathsf{B}})$ to A
- Upon receiving $(\texttt{Input}, n_{\mathsf{A}}, m^{\mathsf{A}}_{1,1}, \ldots, m^{\mathsf{A}}_{n_{\mathsf{A}},X}, l^{\mathsf{A}}_1, \ldots, l^{\mathsf{A}}_{n_{\mathsf{A}}})$ from A, compute $L_j := \mathcal{L}(m^{\mathsf{A}}_{1,j}, \ldots, m^{\mathsf{A}}_{n_{\mathsf{A}},j}, m^{\mathsf{B}}_{1,j}, \ldots, m^{\mathsf{B}}_{n_{\mathsf{B}},j})$ and send $(\texttt{Leakage}, (L_1, \ldots, L_X), (l^{\mathsf{A}}_1, \ldots, l^{\mathsf{A}}_{n_{\mathsf{A}}}))$ to B.
- Upon receiving input $(\texttt{Labels}, l^{\mathsf{B}}_1, \ldots, l^{\mathsf{B}}_{n_{\mathsf{B}}})$ from B, compute the decision tree $\text{tree} = \mathsf{train}(m^{\mathsf{A}}_{1,1}, \ldots, m^{\mathsf{A}}_{n_{\mathsf{A}},X}, m^{\mathsf{B}}_{1,1}, \ldots, m^{\mathsf{B}}_{n_{\mathsf{B}},X}, l^{\mathsf{A}}_1, \ldots, l^{\mathsf{A}}_{n_{\mathsf{A}}}, l^{\mathsf{B}}_1, \ldots, l^{\mathsf{B}}_{n_{\mathsf{B}}})$ and output $(\texttt{Trained}, \text{tree})$ to A

Figure 3.1.: Ideal functionality $\mathcal{F}_{\mathsf{DTTrain}}$.

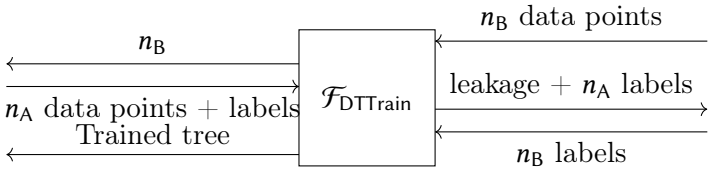

Figure 3.2.: Graphical representation of $\mathcal{F}_{\mathsf{DTTrain}}$. Some details about the concrete values being sent to/from the ideal functionality are omitted.

3.3. Secure Decision Tree Training

In our protocol, we want to train a decision tree without revealing the training data, using the previously constructed Updatable ORE scheme. The core idea is to have training data from one party uORE encrypted under a key which the party itself does not know. To accomplish this, we make use of the updatability of the created ciphertexts. In the second step, we apply the decision tree training algorithm in algorithm 1 to the ORE ciphertexts from the previous step. Here, we make use of the fact that algorithm 1 only requires the comparability of attributes, and is deterministic.

In principle, any decision tree training algorithm with these properties can be used. While the determinism of the training algorithm is required, de-randomization can be done by prepending the protocol with a secure coin-toss and using these coins as input for the decision tree training. If randomness in the training algorithm does not have a security impact, an alternative and more performant way of de-randomization is to use a non-cryptographic PRG with a fixed seed.

Let us first formalize an ideal functionality that captures the security of secure decision tree protocols.

Definition 15 (Ideal functionality $\mathcal{F}_{\mathsf{DTTrain}}$) $\mathcal{F}_{\mathsf{DTTrain}}$ in fig. 3.1 models the security of the decision tree training protocol. A graphical representation thereof is in fig. 3.2. In this setting, there are two parties A and B. The training data has X attributes and discrete labels.

Construction 5 (Protocol π_{DTTrain}) Here, we define the two-party protocol π_{DTTrain} between parties A and B. As before, let X be the number of attributes of the training data. Also, let $m^{\mathsf{A}}_{i,j}$ be the j-th attribute of the i-th training data of A with the labels l^{A}_i (respectively for the dataset of B), and let $(\mathit{Gen}, \mathit{Enc}, \mathit{Cmp}, \mathsf{Upd})$ be an uORE scheme. Then we define the protocol π_{DTTrain} as follows:

B:
- Generate ORE keys $k^{\mathsf{B}}_{i,j}$ for $1 \leq i \leq n_{\mathsf{B}}, 1 \leq j \leq X$
- Send $c^{\mathsf{B}}_{i,j} = \mathit{Enc}(k^{\mathsf{B}}_{i,j}, m^{\mathsf{B}}_{i,j})$ for $1 \leq i \leq n_{\mathsf{B}}, 1 \leq j \leq X$ to A

A:
- Generate ORE keys k^{A}_{j} for $1 \leq j \leq X$
- Send $C^{\mathsf{A}}_{i,j} = \mathit{Enc}(k^{\mathsf{A}}_{j}, m^{\mathsf{A}}_{i,j})$ for $1 \leq i \leq n_{\mathsf{A}}, 1 \leq j \leq X$ to B
- Send labels l^{A}_{i} for $1 \leq i \leq n_{\mathsf{A}}$ to B
- Send $c'^{\mathsf{B}}_{i,j} = \mathsf{Upd}(k^{\mathsf{A}}_{j}, c^{\mathsf{B}}_{i,j})$ for $1 \leq i \leq n_{\mathsf{B}}, 1 \leq j \leq X$ to B

B:
- Compute $C^{\mathsf{B}}_{i,j} = \mathsf{Upd}(1/k^{\mathsf{B}}_{i,j}, c'^{\mathsf{B}}_{i,j})$ for $1 \leq i \leq n_{\mathsf{B}}, 1 \leq j \leq X$
- Train the decision tree on the data points $(C^{\mathsf{A}}_{i}, l^{\mathsf{A}}_{i})_{i=1\ldots,n_{\mathsf{A}}}$ and $(C^{\mathsf{B}}_{i}, l^{\mathsf{B}}_{i})_{i=1\ldots,n_{\mathsf{B}}}$, obtaining a trained decision tree.
- Send the tree to A

A:
- Decrypt and output the decision tree: For each inner node in the tree containing an attribute id j and an encrypted value v, replace v with $\mathit{Dec}(k^{\mathsf{A}}_{j}, v)$.

Note that this protocol has a constant number of rounds, as only three messages are exchanged. Here, A learns no additional information beyond what can be learned from her own training data and the trained decision tree. B receives only the leakage of the ORE scheme of both parties training data for each attribute separately and the respective label. Note that because A uses different keys to encrypt the values of different attributes, B only receives the leakage $\mathcal{L}(m^{\mathsf{A}}_{1,j}, \ldots m^{\mathsf{A}}_{n_{\mathsf{A}},j}, m^{\mathsf{B}}_{1,j}, \ldots m^{\mathsf{B}}_{n_{\mathsf{B}},j})$ for each j. This leakage is much smaller than the entire leakage $\mathcal{L}(m^{\mathsf{A}}_{1,1}, \ldots m^{\mathsf{A}}_{n_{\mathsf{A}},X}, m^{\mathsf{B}}_{1,1}, \ldots m^{\mathsf{B}}_{n_{\mathsf{B}},X})$, because attribute values of different attributes cannot be compared.

Theorem 5 π_{DTTrain} securely realizes $\mathcal{F}_{\mathsf{DTTrain}}$ for static corruption and semi-honest adversaries if the uORE scheme is secure with leakage $\mathcal{L}$.

To show this theorem, we follow the UC framework and construct a simulator, such that the real world running the protocol and ideal world with $\mathcal{F}_{\mathsf{DTTrain}}$ are indistinguishable. Concretely, for any PPT-environment controlling a corrupted party A or B, we show that the environment cannot distinguish between a real interaction of the corrupted party with the uncorrupted one and an interaction of the corrupted party with the ideal functionality through the simulator. Because we consider semi-honest adversaries, the environment chooses the inputs of the honest and corrupted parties and learns their output. It also learns sent and received messages, internal state and randomness of corrupted parties.

We give two different simulators, one for the case where A is corrupted and one for the case where B is corrupted. (If both parties are corrupted, no meaningful security guarantees are left.)

For the case, where A is corrupted, we use the simulator $\mathcal{S}$ from fig. 3.3. To show its validity, we define a number of games G_i, each having (implicitly defined) ideal functionalities $\mathcal{F}_i$ and simulators $\mathcal{S}_i$, and each being a modified version of the previous one. The first game is defined, s.t. the corrupted A interacts with the honest B through the protocol and the last game is the game where the corrupted A interacts with the ideal functionality through the simulator. We show that games G_i are indistinguishable from G_{i+1}.

- When receiving $(\texttt{InputReceived}, n_\mathsf{B})$ from $\mathcal{F}_{\mathsf{DTTrain}}$, invoke the n_B instances of the ORE simulator with the leakage $\emptyset$ (possible by assumption 1) and send the resulting ciphertexts to A.
- When receiving ciphertexts $C^{\mathsf{A}}_{i,j}$, updated ciphertexts $c'^{\mathsf{B}}_{i,j}$ and labels l^{A}_i from A, extract A's ORE keys k^{A}_j and messages $m^{\mathsf{A}}_{i,j}$. Send $(\texttt{Input}, n_\mathsf{A}, m^{\mathsf{A}}_{1,1}, \ldots, m^{\mathsf{A}}_{n_\mathsf{A},X}, l^{\mathsf{A}}_1, \ldots, l^{\mathsf{A}}_{n_\mathsf{A}})$ to $\mathcal{F}_{\mathsf{DTTrain}}$ in the name of A.
- When receiving $(\texttt{Trained}, \text{tree})$ from $\mathcal{F}_{\mathsf{DTTrain}}$ for A, encrypt each value v in a node branching by attribute j as $\mathsf{Enc}(k^{\mathsf{A}}_j, v)$ and send the resulted encrypted tree to A.
- When receiving a query for $\mathcal{F}_{\mathsf{RO}}$, answer consistently or draw a random $x \leftarrow \mathbb{Z}^\times_p$ and return $[x]$ if the message has not previously been queried.

Figure 3.3.: Simulator for our decition tree training protocol when A is corrupted.

- G_0: The execution of π_{DTTrain} with dummy adversary $\mathcal{D}$.
- G_1: The execution with a dummy ideal functionality $\mathcal{F}_1$ that lets the adversary determine all inputs and learn all outputs of the honest party. $\mathcal{S}_1$ is the simulator that executes the protocol π_{DTTrain} honestly on behalf of the honest party using the inputs from $\mathcal{F}_1$ and making outputs through $\mathcal{F}_1$.
- G_2: The same as G_1, except that when sending the encrypted decision tree to A, instead of sending the tree, $\mathcal{S}_2$ extracts the keys k_j from A, uses them to decrypt and reencrypt the tree and sends this reencrypted tree.
- G_3: The same as G_2, except that now $\mathcal{S}_3$ does not perform the training on the ciphertexts, it now performs the training on the plaintext training data, where it extracted A's training data and received B's training data from $\mathcal{F}_4$.
- G_4: The same as G_3, except that instead of sending the values $c_{i,j} = \mathsf{Enc}(k_{i,j}, m_{i,j})$ to A, $\mathcal{S}_4$ generates $c'_{i,j} = \mathcal{S}^{\mathsf{ORE}}(\emptyset)$ and sends these values to A instead.
- G_5: The execution of the ideal functionality $\mathcal{F}_{\mathsf{DTTrain}}$ with the simulator $\mathcal{S}$ as in fig. 3.3.

As we can see, the game G_0 is the game, in which the corrupted party interacts with the honest one using the protocol, whereas G_5 is the game, where the corrupted party interacts with the ideal functionality through the simulator.

Claim 19 $\mathsf{G}_0 \overset{c}{\approx} \mathsf{G}_1$

Proof 19 These changes are only syntactic and therefore oblivious to the environment. The claim follows.

Claim 20 $\mathsf{G}_1 \overset{c}{\approx} \mathsf{G}_2$

Proof 20 The simulator participates in the protocol as the honest party would. Therefore, each inner node of the tree contains elements (j, c), where c is either a ciphertext from A, in which case $c = Enc(k_j^{\mathsf{A}}, m)$ for some m, or a ciphertext from the simulator, in which case

$$c = \mathsf{Upd}(k'^{-1}, \mathsf{Upd}(k_j^{\mathsf{A}}, Enc(k', m))) = Enc(k_j^{\mathsf{A}}, m)$$

for some k' and m. Therefore, decrypting c with k_j^{A} and deterministically reencrypting the result again with the same key results in the same ciphertext. Hence, the games are indistinguishable.

Claim 21 $\mathrm{G}_2 \overset{c}{\approx} \mathrm{G}_3$

Proof 21 It follows from the uORE correctness that evaluating the order on the ciphertexts is equivalent to evaluating the order on the plaintexts (with overwhelming probability), if the ciphertexts to be compared are ciphertexts under the same key. The decision tree training algorithm only compares ciphertexts associated with the same attribute and ciphertexts for an attribute j are all ciphertexts under key k_j^{A} (either directly or by updating and reverse-updating). Therefore, it follows from the correctness of the ORE scheme that encrypting the messages, training the decision tree on the ciphertexts and decrypting the decision tree again (as done in G_2) results (with overwhelming probability) in the same tree as directly training the decision tree on the plaintexts (as done in G_3).

Claim 22 $\mathrm{G}_3 \overset{c}{\approx} \mathrm{G}_4$

Proof 22 This change is only relevant for the ciphertexts the simulator sends to A. While it receives these ciphertexts back, updated with A's keys, they are no longer used by the simulator in any further computations. Because of assumption 1 and the fact that the keys $k_{i,j}$ are only ever used for encryption once, it holds that for all messages m, including the messages the simulator encrypts in G, that $\mathcal{L}(m) = \emptyset$. Therefore, an adversary that can distinguish between G_3 and G_4 can be used to distinguish between $\mathsf{REAL}^{\mathsf{ORE}}_{\mathcal{A}}(\kappa)$ and $\mathsf{SIM}^{\mathsf{ORE}}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\kappa)$ using a hybrid argument.

Claim 23 $\mathrm{G}_4 \overset{c}{\approx} \mathrm{G}_5$

Proof 23 The only difference between these two games is who performs the training of the decision tree. In G_5, the simulator $\mathcal{S}_5$ performs the training of the decision tree, while in G_6, the ideal functionality $\mathcal{F}_6$ performs the training. In both cases, the training is performed on the same training data and as training is deterministic, it results in the same decision tree in both cases.

For the case, where B is corrupted, we use the simulator $\mathcal{S}$ from fig. 3.4. Again, we define hybrid games to show its validity:

- G'_0: The execution of π_{DTTrain} with dummy adversary $\mathcal{D}$.
- G'_1: The execution with an ideal functionality $\mathcal{F}'_1$ that lets the adversary determine all inputs and learn all outputs. $\mathcal{S}'_1$ is the simulator that executes the protocol π_{DTTrain} honestly on behalf of the honest party using the inputs from $\mathcal{F}'_1$ and making outputs through $\mathcal{F}'_1$.

- When receiving ciphertexts $c^{\mathsf{B}}_{i,j}$ from B, extract the messages $m^{\mathsf{B}}_{i,j}$ from B and send $(\texttt{Input}, n_{\mathsf{B}}, m^{\mathsf{B}}_{1,1}, \ldots m^{\mathsf{B}}_{n_{\mathsf{B}},X})$ to $\mathcal{F}_{\mathsf{DTTrain}}$ in the name of B.
- Upon receiving $(\texttt{Leakage}, (L_1, \ldots, L_X), (l^{\mathsf{A}}_1, \ldots, l^{\mathsf{A}}_{n_{\mathsf{A}}}))$ from $\mathcal{F}_{\mathsf{DTTrain}}$ for B, proceed as follows for all j:
  - For each i, compute the sub-leakage $L^{\leq i}_j$ from L_j. This is possible using assumption 2.
  - Generate ciphertexts $C_{i,j} = \mathcal{S}^{\mathsf{ORE}}(L^{\leq i}_j)$ for $1 \leq i \leq n_{\mathsf{A}} + n_{\mathsf{B}}$
  - Set $c'^{\mathsf{B}}_{i,j} \leftarrow \mathsf{Upd}(k^{\mathsf{B}}_{i,j}, C_{i+n_{\mathsf{A}},j})$ for $1 \leq i \leq n_{\mathsf{B}}$

  Send the $C_{i,j}$, $c'^{\mathsf{B}}_{i,j}$ and l^{A}_j to B.
- Upon receiving tree from B, extract labels l^{B}_j from B and send $(\texttt{Labels}, l^{\mathsf{B}}_1, \ldots, l^{\mathsf{B}}_{n_{\mathsf{B}}})$ to $\mathcal{F}_{\mathsf{DTTrain}}$ in the name of B.

Figure 3.4.: Simulator for our decition tree training protocol when B is corrupted.

- G'_2: The same as G'_1, except that when $\mathcal{S}'_2$ would update a ciphertext, which B computed as $ct = \mathsf{Enc}(k', m)$ to $ct^* = \mathsf{Upd}(k, ct)$, instead it computes $ct^* = \mathsf{Upd}(k', \mathsf{Enc}(k, m))$ using the corresponding key k' and message m it extracts from B.
- G'_3: The same as G'_2, but instead of decrypting and outputting the tree received from B, the simulator extracts the training data from B. Then it uses both parties training data and labels to train the decision tree in plain text and outputs the tree to A through $\mathcal{F}'_3$.
- G'_4: The same as G'_3, except that whenever we would perform an encryption (both in the encrypt- or update-step) as $\mathsf{Enc}(k, m)$, we compute it with $\mathcal{S}^{\mathsf{ORE}}$ using the corresponding leakage, using one instance per k.
- G'_5: The ideal functionality $\mathcal{F}_{\mathsf{DTTrain}}$ with the simulator $\mathcal{S}$.

Now we look at the case, where B is corrupted:

Claim 24 $\mathsf{G}'_0 \overset{c}{\approx} \mathsf{G}'_1$

Proof 24 These changes are only syntactic and therefore oblivious to the environment. The claim follows.

Claim 25 $\mathsf{G}'_1 \overset{c}{\approx} \mathsf{G}'_2$

Proof 25 As the difference between these two games is the order in which *Enc* and Upd are applied, this claim follows directly from the updatability property.

Claim 26 $\mathsf{G}'_2 \overset{c}{\approx} \mathsf{G}'_3$

Proof 26 The only difference between these two games is the decision tree that is oututted to A. In G_3, the simulator knows A's training data, because it received them from $\mathcal{F}_3$ and it knows B's training data, because it can extract them from B. From the uORE correctness and the fact that the decision tree training is deterministic, it follows that the decision tree $\mathcal{S}_3'$ computes in plaintext is identical to the one B computes in G_3' and G_2' on the ciphertexts.

Claim 27 $\mathsf{G}_3' \overset{c}{\approx} \mathsf{G}_4'$

Proof 27 In G_3', the ORE encryption is performed using real keys k_j, whereas in G_4', the ORE simulator is used to generate ciphertexts. To show this indistinguishability, we define additional hybrids H_j, where for the first j attributes, the ORE simulator is used to generate ciphertexts, and for all other attributes, a real encryption is performed. It holds that $\mathsf{G}_4' = \mathsf{H}_0$ and $\mathsf{G}_5' = \mathsf{H}_X$.

The indistinguishability of these games therefore follows from the ORE-security using a standard hybrid argument.

Claim 28 $\mathsf{G}_4' \overset{c}{\approx} \mathsf{G}_5'$

Proof 28 The difference between these two games is that in G_5', the simulator performs the training and sends the trained decision tree to A through the ideal functionality, while in G_6' the ideal functionality performs the training and outputs it directly. As the decision tree training algorithm is deterministic and the same input to the training process is used by $\mathcal{S}_5'$ and $\mathcal{F}_6'$, the tree is identical and the games are indistinguishable.

The security of the protocol π_{DTTrain} (theorem 5) now follows from claim 19 – 23 and claim 24 – 28.

3.3.1. Variations of the Training Process

In the protocol from construction 5, we only considered standard decision tree training. When using decision trees in practice, additional steps are usually taken like pruning, limiting the depth of the tree, gradient boosted training or training a decision forest for better classification accuracy.

Performing gradient boosted training is inherently not compatible with our approach, as it requires performing arithmetic operations on attribute values, which ORE schemes do not support.

Pruning and limiting the depth of the tree could be performed by B by adding a leaf node instead of an inner node to the tree, whenever the number of datapoints at the current position in the tree is small enough or if a certain depth of the tree is reached during the training process. Both of these techniques are compatible with our decision tree training protocol. Indeed any form of pruning that adheres to the limitation that attributes can only be compared, but no arithmetic can be performed on the attribute values, is compatible with our approach.

Additionally, our protocol can be used for training a decision forest. Training a T-tree decision forest can be done as follows:

1. Each party partitions their training data into T subsets.

2. The parties run π_{DTTrain} once for each subset in parallel to perform the training on each data.
3. After receiving the T decision trees from B, A outputs the decision forest containing these trees.

This realizes a forest-variant of the ideal functionality $\mathcal{F}_{\mathsf{DTTrain}}$. The security of the protocol follows from the universal composability theorem in the UC model.

Another variation is to consider training data that is split vertically between parties, i.e. A has one part of the attributes and B has a different set of the attributes of the same dataset (and they share some kind of id attribute to match up the data). In this setting, training is easily possible with the ORE-based approach by having A encrypt her attribute values using an ORE scheme with one key per attribute and sending the ciphertexts to B. B can then train a decision tree using A's encrypted attributes and his own attributes in plaintext, as no comparison between his and A's data is required. Indeed this does not even require the ORE scheme to be updatable.

3.3.2. Graceful Degradation using Enclaves

Hardware enclaves allow other parties to verify the code running in them while ideally hiding all internal state. While hiding internal state is difficult due to side channels, either inherent in the enclave program (like timing or memory access patterns), or due to the used platform (like power consumption, cache timing or other microarchitecural state side channels), the minimal functionality of an enclave, namely to attest the correct program execution, is usually well hardened and implemented side-channel free. These side channels have motivated two major security models for the functionality provided by an enclave system: "regular" enclaves (hiding all internal state) and transparent enclaves (revealing all internal state, especially the used randomness, but not attestation keys). If additionally attestation keys are leaked, the enclave provides no meaningful security. In between transparent and regular, enclave models with varying amount of side channels can be useful, like explicitly modeling a memory access side channel, as done in [86].

In the minimal form of a transparent enclave, they can be seen as a generic passive-to-active compiler. Executing an arbitrary passively secure protocol inside an enclave allows other involved parties to verify the produced attestation evidence to ensure that the other parties executed their part of the protocol correctly.

When applied to the presented decision tree training protocol (construction 5), we can go one step further: When the protocol for B is executed inside a non-transparent enclave, the leakage from the ORE scheme to B is hidden. To implement this, A needs to send her ciphertexts to the enclave of B confidentially, which can be done by performing a key exchange into the enclave. Additional care needs to be taken to ensure that the enclave program does not leak more information about the input than necessary. Also, the enclave needs to hold a significant amount of memory to store the ORE-encrypted training data. Requiring access to this amount of data will result in many memory-management operations by the enclave system and thus has an impact on the overall performance.

An overview of the provided security in the different scenarios is given in Table 3.1. The algorithms that are compared are:

- the proposed algorithm with MSD radix sort inside an enclave

Table 3.1.: Comparison of security of our decision tree training protocol, with and without radix sort, under different enclave security assumptions.

	ours with radix	ours without radix	plaintext with radix	plaintext without radix
fully secure enclaves	✓	✓	✓	✓
with memory side channels	ORE leakage	ideal-ORE leakage	ORE-leakage	ideal-ORE leakage
transparent enclaves	ORE leakage	ORE leakage	full leakage	full leakage

- the proposed algorithm with comparison-based sorting, where all comparisons are done memory-oblivious (e.g. using the primitives from [86])
- the plaintext decision tree training algorithm executed inside an enclave using MSD radix sort
- the plaintext decision tree training algorithm executed inside an enclave with comparison-based sorting using memory-oblivious comparisons

When using a fully secure enclave, all computation happens inside the enclave and cannot be eavesdropped or tampered with, so all algorithms are secure. When we assume memory side channels, radix sort exploits the concrete ORE leakage and thereby leaks it via the memory access patterns, both in the plaintext and encrypted variant. The other two algorithms only use the result of pairwise comparisons and thereby leak at most an ideal-ORE leakage (cf. section 1.3.5). Using comparison-based sorting and memory-oblivious comparisons, memory access patterns can only leak at most the results of those comparisons. When the enclave becomes fully transparent, all information stored inside the enclave is potentially leaked, which are ORE ciphertexts for the first two algorithms and the plain training data for the last two.

As can be seen, our algorithm provides a more graceful degradation of security when the enclave fails to provide its security promise (e.g. due to expected or unexpected side channels in the implementation), at the cost of higher computational overhead. Therefore, if the trust model of the enclave is uncertain, combining our approach with secure enclaves allows decision tree training with less leakage than solely relying on either approach exclusively.

Note that the plaintext algorithm needs to employ similar techniques to the decision tree evaluation algorithm from [86] to avoid additional leakage. However as the authors only describe an evaluation but no training algorithm, no direct comparison can be made with the decision tree algorithm they provide.

3.4. Analysis of the Leakage

Grubbs et al. [51] have shown that in the context of encrypted databases, there are datasets, such that when encrypting them using ORE schemes, no meaningful security guarantees are left. They have shown that in one of their datasets containing first and last names, they could recover 98% of all first names and 75% of all last names in a database encrypted using the scheme from [35].

While these results also apply to our scheme and therefore also to our decision tree training protocol, we want to emphasize that these results are not universally applicable to all datasets. In their example, the dataset of first names had relatively low entropy with the most common first name appearing in 5% of all cases. Jurado, Palamidessi, and Smith [62] have shown for example

Table 3.2.: Proportion of leaked bits to total bits, for leakage $\mathcal{L}$ on n uniformly random messages of length k bits.

	$k = 8, n = 8$	$k = 8, n = 256$	$k = 8, n = 512$	$k = 64, n = 1000$	$k = 64, n = 10000$	$k = 64, n = 50000$
Leaked Bits	39.1%	93.6%	97.7%	16.0%	21.3%	24.9%

that attacks recovering all ORE-encrypted data is possible when the amount of ciphertexts is large compared to the message space, whereas this is not possible if the amount of ciphertexts is significantly smaller than the message space.

To give meaningful security guarantees, we analyze the leakage from an information-theoretic point of view. We consider the uniform distribution of messages, because for this distribution, each bit of the message space contains one bit of information. In a first step, we analyze the leakage when both parties use uniformly random training data. In a second step, we give an upper bound for the leakage, when an adversary (B in the case of the decision tree training protocol) selects its training data maliciously. Finally, we argue why considering only the uniform distribution is sufficient.

3.4.1. Leakage for Random Message Selection

In the case where both parties use uniformly random messages as inputs, we can experimentally estimate the information leaked on different datasets. We consider a bit to be leaked, if the leakage function allows to infer the value of this bit. In our experiments, we sample n data samples uniformly at random from the bitstrings of length k and count the number of leaked bits. These results are available in table 3.2. As we can see, the experiments show a significant leakage when the amount of data samples is large compared to the domain, leaking nearly all bits if there are more data samples than there are possible messages in the domain. If the message space is significantly larger than the amount of messages leaking information, the amount of bits leaked is less than 25% (for messages chosen uniformly at random). This matches the result of Jurado, Palamidessi, and Smith [62], namely that for ORE to provide a benefit over comparing the messages in plain text, the message space must be much larger than the amount of messages encrypted under the same key.

This suggests that the leakage from our ORE approach is small enough to provide some security if the attributes of the training data have sufficiently large entropy, compared to the amount of training data. An example for attributes that have naturally high entropy are geopositions with latitude and longitude.

3.4.2. Additional Leakage for Malicious Message Selection

We also want to give an analytical upper bound of the additionally leaked information for independently uniformly distributed training data if one party selects their inputs maliciously. This is not a classical setting for ORE, but makes sense in our case, because here, the party receiving the leakage (B in the case of π_{DTTrain}) contributes data influencing the leakage.

Let $\mathcal{L}$ be the leakage function as in theorem 2. Consider the following experiment with an attacker choosing $N = 2^k$ messages:

1. The adversary chooses N messages $m_i^* \in \mathcal{M} = \{0,1\}^n$.

2. The experiment chooses a message $m \leftarrow \mathcal{M}$ uniformly at random.
3. The adversary learns $\mathcal{L}(m, m_1^*, \ldots, m_N^*)$.

Choosing the messages optimally to have the maximum (over the attacker-chosen $m_1^*, \ldots, m_N^*$) leaked information on average (over m), the adversary receives no more than $(k+2)$bit of information.

Consider the first attacker-chosen message m_1^*. The leakage contains the information whether the first bit of m is equal to the first bit of m_1^* (the position of the most significant different bit is $\mathsf{hsb}(m \oplus m_1^*) > 1$) or if they are unequal ($\mathsf{hsb}(m \oplus m_1^*) = 1$). Therefore, the attacker learns the first bit of m containing 1bit of information. The same argument also holds for the second bit, but only if the first bit is the same in m and m_1^*.

Therefore, he obtains the second bit (and therefore one additional bit of information) with probability 1/2, as m's first bit was chosen uniformly at random. Generalizing this for more bits, he learns the i-th bit of m with probability $1/2^{i-1}$. For the expected information, we get the geometric series:

$$\sum_{i=1}^{n} \frac{1}{2^{i-1}} \cdot 1\text{bit} \leq 2\text{bit}.$$

To generalize the maximum leakage to $N = 2^k$ attacker-chosen messages, we consider the information stored in the first $k+1$ bits of m and the remaining bits separately. (Here, we assume $k+1 < n$, as otherwise, the attacker would choose every second message from $\mathcal{M} = \{0,1\}^n$ and learn the content of all ciphertexts.) The first $k+1$ bits of m contain $(k+1)$bit of information, so that is the maximum amount of information an attacker can infer. For the remaining message bits, the probability of m having the same $k+1$-bit prefix as any of the m_i^*, and therefore causing a bit to leak, is $1/2^{k+1}$. Ignoring possible overlap between messages gives us an upper bound of

$$2^k \cdot \frac{1}{2^{k+1}} \cdot 2\text{bit} = 1\text{bit},$$

where the 2bit is again an upper bound for the geometric series over the expected leaked information in the remaining bits.

Combining the two leakages, we get $(k+1)$bit + 1bit = $(k+2)$bit. This establishes an upper bound for the average leakage.

While we only considered a single message m selected by the experiment, it can naturally be extended to a setting, where the experiment chooses multiple messages $m_1, \ldots, m_{N'}$ and the adversary receives $\mathcal{L}(m_1, m_1^*, \ldots, m_N^*), \ldots, \mathcal{L}(m_{N'}, m_1^*, \ldots, m_N^*)$ instead. (We are only interested in the additional leakage by the attacker, the leakage $\mathcal{L}(m_1, \ldots, m_{N'})$ is already analyzed in the previous subsection.) As long as these messages are chosen independently from each other, the maximum leakage per message m_i remains the same. This multi-message scenario captures the leakage B receives about A's training data in π_{DTTrain} for each attribute when selecting his messages maliciously.

Note that this analysis extends to the use of multiple uncorrelated attributes, due to the use of separate ORE keys per attribute. If multiple attributes are correlated, this correlation can be interpreted as information known in advance by the adversary. Hence, the upper bound on the average leakage also holds for the information given what is known to the adversary, due to a union bound.

Table 3.3.: Operations per second using ORE with a 32-bit message space. Sort refers sorting 1000 ciphertexts using Java's Arrays.sort() or our own implementation of MSD radix sort.

	Our scheme	Scheme from [35]
Encryption	$1.6 \cdot 10^2$	$7.2 \cdot 10^4$
Update	$2.0 \cdot 10^2$	–
Comparison	$7.0 \cdot 10^4$	$6.9 \cdot 10^7$
Sort	$8.0 \cdot 10^0$	$3.8 \cdot 10^5$
Radix sort	$6.9 \cdot 10^1$	$1.0 \cdot 10^3$

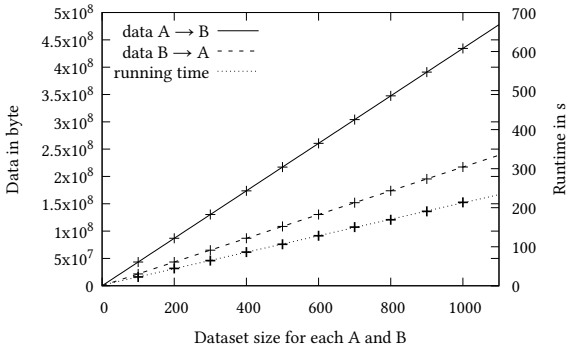

Figure 3.5.: Full results of the benchmarks from table 3.4 where the datasets of A and B are of the same size.

Overall, for any adversarially selected input data of B in π_{DTTrain} consisting of $N = 2^k$ training data, the expected leaked information consists of $(\log_2(N) + 2)$bit per A's training data per attribute.

3.4.3. Transformation for Non-uniform Distributions

While the distribution of input data is important because changing the encoding of messages (and therefore also the distribution over the message space) influences how much and what information is leaked, we want to emphasize that this is not a restriction, as any input data distribution can be encoded in such a way that the resulting distribution is uniform. We now sketch one way to do this.

Let μ be the probability measure of the distribution over the message space $\mathcal{M}$. Assume there exists a l, such that $\mu(m) \cdot 2^l \in \mathbb{N}$ holds for all $m \in \mathcal{M}$.[1] Then we encode the messages into $\{0,1\}^l$ as follows: The valid encodings of a message m are the $\mu(m) \cdot 2^l$ bitstrings starting from the bitstring of the binary representation of $\mu(\{m' \in \mathcal{M} \mid m' < m\}) \cdot 2^l$. To encode a message, one of its valid encodings is used uniformly at random. This encoding preserves the order, but the resulting distribution is uniform over $\{0,1\}^l$.

3.5. Implementation and evaluation

We implemented the ORE scheme from [35], our updatable ORE scheme from construction 4, as well as our decision tree training protocol from construction 5 in Java/Kotlin[2] and used it to evaluate the practical efficiency of our protocol. For the group $\mathbb{G}$, we used the Ed25519 curve. We used the PRF from eq. (1.1), implemented the random oracle using SHA-256 and mapped its output to points on the curve $\mathbb{G} \setminus \{[0]\}$. We ran our experiments on a machine with two AMD EPYC-Rome 7282 processors with 16 Cores/32 Threads and 90 GB of RAM.

[1] If the assumption does not hold, an approximation of μ with powers of 2^{-l} for some l results in a distribution, that is computationally indistinguishable from uniform.

[2] https://github.com/kastel-security/ORE-Decision-Tree

3.5.1. Evaluation of the Updatable ORE Scheme

For completeness, we start by comparing our updatable ORE scheme with the non-updatable scheme from Chenette et al. [35]. The benchmark results can be seen in table 3.3.

For encryption, our scheme is two orders of magnitude slower, because they only need to evaluate a PRF, while evaluating the key-homomorphic PRF in our case requires a scalar multiplication in the group. Updating a ciphertext is roughly as fast as encryption with the running time of both operations being dominated by the scalar multiplication.

Comparisons using our updatable scheme are three orders of magnitude slower than with the non-updatable scheme because the non-updatable scheme requires no cryptographic operations, while we require one group operation per comparison. This translates over to comparison-based sorting. Radix sort offers a running time improvement of one order of magnitude, because – in contrast to comparison-based sorting, where one group operation is required per comparison per element – only one group operation is required per element in total. Radix sort does not offer a performance improvement with the non-updatable scheme, because the running time is not dominated by comparing the elements (or bits thereof), but by the sorting algorithm itself.

3.5.2. Evaluation of the Protocol

For our experiments of the training protocol, we used the unmodified MNIST dataset using all 784 attributes and datapoints of all 10 labels. We also ran experiments on the Boston Housing and Titanic dataset[3], which we modified to ignore entries with null-attributes, as well as discrete attributes. As required for decision tree training, we also discretized the labels. This leaves seven attributes in the Boston Housing dataset and five attributes in the Titanic dataset.

In the MNIST dataset, attribute values are 8 bit unsigned integers. In the other datasets, we converted all attributes to integers by taking their 32 bit IEEE 754 representation, reinterpreting it as a 32 bit unsigned integer and dropping the last bit, therefore obtaining an unsigned 31-bit integer, which preserves the order of all positive floating point numbers, including "+0". We used the first n_{A} datapoints as training data for A and the last n_{B} datapoints as training data for B. We used the training algorithm in algorithm 1 with the Information Gain heuristic. As the training algorithm used here is the same as for plaintext training, the accuracy of the trained model is identical to a model trained on the same data in plaintext. The results can be seen in table 3.4 and fig. 3.5.

As we can see, the protocol is viable, both computationally, as well as from a network traffic perspective. We can also see that the effects of the dataset size of A are less significant than the effects of the dataset size of B, as B's data needs to be processed three times – when encrypting, when updating, when reverse updating – whereas A's data only needs to be processed once during encryption. A similar argument holds for the network traffic. While we did run the experiments on the same machine, we expect the performance to only differ insignificantly from our test results when run on separate machines over a LAN or WAN. This is because we only have three rounds of interaction, so the effects of network latency are insignificant. Additionally, the total network traffic is well below the limits of a normal internet uplink.

[3] The datasets are available on https://www.kaggle.com/.

Table 3.4.: Benchmark results of the protocol on the MNIST dataset and modified versions of the Boston Housing and Titanic datasets, and a custom dataset comparable to the one used by [53], with and without 50 ms of network latency.

Dataset	#Attributes	Compute Threads per party	Dataset Size A	Dataset Size B	Computation time	Network traffic A ← B	Network traffic A → B
MNIST	784	1	100	100	243.9 s	21.7 MB	43.4 MB
		16	100	100	22.6 s	21.7 MB	43.4 MB
		16	500	500	106.3 s	108.6 MB	217.2 MB
		16	500	1000	180.6 s	217.2 MB	325.8 MB
		16	1000	500	135.1 s	108.6 MB	325.8 MB
		16	1000	1000	213.6 s	217.2 MB	434.3 MB
Boston Housing	7	1	253	253	23.2 s	1.9 MB	3.6 MB
		16	253	253	3.7 s	1.9 MB	3.6 MB
Titanic	5	1	357	357	23.8 s	2.1 MB	3.6 MB
		16	357	357	5.0 s	2.1 MB	3.6 MB
Custom	7	16	4096	4096	20.1 s	8.8 MB	15.9 MB
Custom (with latency)		16	4096	4096	20.2 s	8.8 MB	15.9 MB

Running time comparison with Hamada et al. [53] To compare our results with the results of [53], the current state-of-the-art in MPC-based decision tree training, we generate a dataset consisting of 2^{13} samples with 11 attributes each, which results in a trained decision tree of depth 42. On this dataset, our protocol takes 20.1 s. In [53], they have performed a benchmark on a dataset of the same size and amount of attributes and a tree depth of 40 on a machine comparable to ours. In this scenario, their protocol takes 43.61 s, which is slower than our protocol by a factor of ≈ 2.

When adding 50 ms of artificial network latency, the running time of their protocol increases to 4821.56 s, which is caused by the many rounds of interaction in their protocol. In contrast to this, when adding the same artificial network latency to our protocol, the running time does not change noticeably, still only requiring 20.2 s, because our protocol only consists of three rounds. In this scenario, our protocol is faster by several orders of magnitude.

Running time comparison with Abspoel, Escudero, and Volgushev [2] In [2], the authors did not measure the running time of the entire decision tree training algorithm, but only extrapolated its runtime based on benchmarks of its basic operations. To compare the runtime performance of their approach with ours, we use their extrapolation formula for their runtime to a setting, for which we have have benchmark results with our approach. They use

$$T(N, m, \Delta) \approx m \cdot (S(N) + (2^{\Delta} - 1)I(N) + 2^{\Delta}L(N))$$

to estimate their runtime, where N, m and Δ are the number of training data, the amount of attributes and the maximum depth of the decision tree and $S(N), I(N)$ and $L(N)$ are the time for sorting, and computing an inner or leaf node on N training data points.

We use this formula to estimate the runtime of their approach to the titanic dataset, where $N = 357, m = 5$ and the depth of the resulting decision tree is $\Delta = 25$. Using the optimistic values $S(256) = 0.392$ s, $I(256) = 0.127$ s and $L(256) = 0.004$ s, we obtain a runtime estimate of $\approx 2.2 \cdot 10^7$ s in the passively secure setting. This is several orders of magnitude slower than with

our approach, which is mostly caused by their training algorithm having exponential runtime in the tree depth.

Limiting the depth of the decision tree to $\Delta = 10$ only affects the runtime of our approach insignificantly, as the majority of the runtime comes from encrypting and updating the training data. Their approach, however, is significantly sped up by this change, estimated to only have a runtime of ≈ 672 s, which is still significantly slower than our approach.

Applicability comparison with [53, 2] While our protocol solely relies on Order-Revealing Encryption, both of the protocols from [53, 2] are built on top of generic MPC primitives. Therefore, future advances in (u)ORE or general purpose MPC respectively, will lead to performance improvements of these protocols.

Due to the asymmetric nature of our protocol, it is fixed for the two-party setting with one passive corruption. In addition to this setting, Hamada et al. [53] and Abspoel, Escudero, and Volgushev [2] state that their protocols can fulfill different trust models, such as two-out-of-three corruptions, if the underlying protocol for these MPC primitives is chosen accordingly, however this may cause additional computational/network overhead.

While their protocol can only be applied to the training algorithms they consider, our protocol can generically use any decision tree training algorithm that adheres to the limitations of being deterministic and only requiring comparisons on the training data. Indeed this is even covered by our security proof.

3.6. Conclusion

We have constructed an Updatable Order-Revealing Encryption scheme, which allows to update ciphertexts from one key to another using a key-homomorphic PRF. This construction is secure under the same leakage function as established ORE schemes, leaking the order of the encrypted messages, as well as the position of the most significant bit in which they differ.

Using such an Updatable ORE scheme, we have constructed a passively secure protocol that allows for securely training a decision tree on two parties' inputs without revealing the inputs to the other party. This protocol can either be used by itself or can be used as a building block to train a decision forest.

We have experimentally verified this decision tree protocol and are able to compute a decision tree on the Titanic dataset, equally partitioned between both parties, within 5.0 seconds. The experiments have also shown that this approach is faster than the current state-of-the-art approaches [2, 53] and orders of magnitude faster when considering network latency or training high-depth decision trees. However, this speedup comes at the cost of some information leakage.

Analyzing the leakage of the ORE scheme, we have found that while it is significant, it is also hides a large proportion of the training data. This provides us with an interesting trade-off between security and efficiency: We leak more information but are faster than relying entirely on MPC to train a decision tree, but we are more secure, but less efficient than performing training in plaintext. We have also found that the proportional information leaked is larger on low-entropy attributes than on high entropy attributes. Whether this leakage is acceptable needs to be decided for each usecase individually. To further reduce the leakage, we show how

this approach can also be used in a secure enclave, reducing the leakage even more in a graceful degradation manner, even in the presence of low-entropy attributes.

This leaves us with a special-purpose protocol for decision tree learning that is more performant than generic solutions in a scenario that fits its limitations:

- The protocol has some information leakage and can only be used in a scenario, in which this is acceptable.
- The concrete algorithm for training needs to fulfill some constraints:
  - The training algorithm needs to be deterministic.
  - The split heuristic needs to be evaluated based on comparisons only.
  - As described the protocol allows training between exactly two parties.
  - For training forests: The partitioning of each party's data needs to be independent of the data of the other party.
- The protocol requires more computation compared to the number of communication rounds, so its strength shows better in a higher-latency setting.

While some leakage is inherent with this approach, this also leads us to two interesting open questions:

- How to construct an Updatable ORE scheme with a smaller leakage or in the left-right framework of [70]?
- How to devise an efficient actively-secure protocol based on Updatable ORE schemes?

4. Even Less Hardware Trust

This chapter is based on joint work with Marco Liebel, Jeremias Mechler and Jörn Müller-Quade[41]. In this work I contibuted the core idea of separating randomness and non-determinism in the execution, developed the key insight, how non-determinism in the real execution should be treated formally, and provided the base for building a practical system that can be tested and benchmarked. Realizing a such a system and executing benchmarks was the topic of Marco Liebel's Master Thesis[72].

4.1. Introduction

When using untrusted systems, even basic properties such as correctness are not guaranteed: A malicious system may change the execution in an arbitrary way, possibly using different inputs or returning wrong results. Depending on the performed task, the consequences of a deviation may be grave, including a total loss of privacy.

Apart from such crass deviations in an execution, the influence of a malicious system may be much more subtle, but the consequences may be just as severe: In today's systems, privacy is often achieved by relying on cryptographic building blocks, for example for encryption or key exchange. Often, the security of these building blocks relies on a) high-quality randomness and b) the secrecy of cryptographic keys. On an untrusted system, neither may be guaranteed. For example, randomness provided by the system may be of low quality, e.g. with low entropy, leading to a total loss of privacy. Also, a malicious system could leak sensitive information via side-channels, for example via the timing of network packets or the choice of sequence numbers[1].

Interestingly, even advanced and expensive techniques such as secure multi-party computation [50, 105] may not offer protection in this setting: Many established security notions just distinguish between honest and corrupted parties and only provide security guarantees for parties that are honest. If an honest party (or its system) gets corrupted, no security guarantees are given anymore. This is not only a definitional artifact, but a consequence of the system participating in a secure computation typically holding the input and being responsible for returning the output.

Trusted Computing. Towards protecting against corrupted systems, a popular technique that continues to see widespread use is so-called "trusted computing". By embedding specific modules that act as "roots of trust" into the system and/or its individual components, the goal is to a) provide the ability to verify that the execution has not been tampered with and b) possibly also reduce the attack surface, for example in the case of secure enclaves. For example, trusted

[1] https://github.com/Kicksecure/tirdad

computing may allow to verify that the executed software is the intended one (or preventing a system from running "unwanted" software completely) before giving sensitive input. Practical implementations of this system use a Trusted Platform Module (TPM) [32], or focus on securing part of the execution like Intel Software Guard Extensions (SGX) [38] for specific programs and AMD Secure Encrypted Virtualization (AMD SEV) for individual virtual machines. However, the hardware and software components that enable trusted computing in the first place (including the potentially large trusted computing base) are usually very complex, often closed-source and proprietary, making their honesty and correctness a very strong assumption that may not be justified in practice. Indeed, if these components are malicious or faulty, for example because the manufacturer is forced by a government agency to embed a backdoor, no security may be provided at all.

A Different Approach. This work explores an orthogonal approach for securing a computation, namely by distributing the trust between two machines instead of focusing it on a single one. By setting up a computation with CoRReCt, only one out of two general-purpose systems needs to be trusted. These systems may come from different manufacturers and run different operating systems. This is in contrast to recent trusted computing techniques such as AMD SEV, Intel SGX or Intel TDX, which are available from one (processor) manufacturer only and currently may have limited operating system support. To this end, we adapt a well-known approach used to increase fault tolerance, namely the parallel execution on multiple heterogeneous systems, whose results are compared (or have a majority vote performed) by a simple application-agnostic comparator. In such a setting, a (not necessarily strict) majority of correct or honest systems leads to a correct output or no output at all, but never to a wrong output.

A challenge in this setting is non-determinism that may occur in the execution of complex systems, for example due to timing. In the presence of such non-determinism, performing the same computation on multiple (and even honest) systems may unintentionally lead to different results. To deal with this problem, an established technique is to record non-deterministic events in one execution and to replay them in the other execution(s). This requires either very similar systems [95] or an appropriate level of abstraction.

However, when considering security and, in particular privacy, this naïve approach may be insufficient: For many tasks, high-quality randomness is necessary. However, randomness is often obtained by harvesting non-deterministic events[2]. In a setting where non-determinism may be controlled (to some extent) by an adversary, the quality of the randomness is no longer guaranteed, even if the replaying system is honest. At the same time, using independent randomness on each system is impossible, as this might lead to inconsistent executions with different results.

In our approach, which we call CoRReCt, short for compute, record, replay, compare, we deal with this problem by obtaining cryptographically secure randomness via a special protocol in a precomputation phase. This randomness can then be used for the subsequent computation, greatly reducing an adversary's influence from exploiting non-deterministic behavior. In case of the Linux kernel, we make sure that only secure randomness is used, as added non-determinism can reduce entropy from the point of view of an adversary. For our implementation, we modified the Linux kernel to only use randomness created through a cryptographic protocol.

[2] https://github.com/torvalds/linux/blob/994d5c58e50e91bb02c7be4a91d5186292a895c8/drivers/char/random.c#L1127

We demonstrate the usefulness of our approach with the use-cases of i) secure multi-party computations, ii) secure messaging using Tinfoil chat [87] and iii) software building.

4.1.1. Related Work

Redundancy. Duplicating components for fault tolerance is a general approach to increase the reliability against accidental failure. This practice is also applied for general computations, for example in ARM Cortex-R realtime cores [60].

Byzantine Fault Tolerance. A related problem, considered by byzantine fault tolerance (BFT) [68, 31], is to achieve consensus between multiple systems where a subset of the systems may arbitrarily misbehave. In particular, this means corrupted systems might send differing information to honest systems. For performance reasons, CoRReCt does not intend to achieve consensus, but merely to detect byzantine faults in an execution with two machines. This allows CoRReCt to establish guarantees for the case that no faults occur, but the execution is still influenced by an adversary through allowed behavior.

Record-Replay. Record-replay systems have been investigated for some time, with debugging of non-deterministic applications as their main focus [94, 97]. Recording non-determinism and thereby enabling deterministic replay allows for effective debugging. It provides features like reverse stepping to help understand complex bugs, especially those of non-deterministic nature.

The use of recording and replaying non-deterministic behavior of VMs for fault tolerance in case of hardware failure has been explored for a longer time, with early research implementations [18] and newer, practical implementations [95, 106, 74], some of which are production-ready today. Although replaying to enable fault tolerance seems quite similar to our goal, the main objective is to continue execution of a virtual machine (VM) in case one VM host goes offline. That means that in practice, only one of the machines will produce outputs, until a fail-over is decided and the replaying VM takes over. As a consequence, replaying for fault tolerance is usually done on similar hardware, so VMs remain in sync, while still being able to use hardware support for virtualization. As outputs are not compared, these previous approaches do not protect against malicious output produced by a corrupted system. In contrast, our approach relies on multiple hosts, possibly from different vendors, cross-checking the computation results to prevent one host from manipulating them.

The canonical alternative approach for handling non-deterministic behavior, is to remove it. That might be difficult in all cases, but there is work[84], that removes the majority of non-deterministic behavior.

Robust Combiners. Robust combiners allow to combine different implementations of (mainly cryptographic) primitives such that even if an unknown subset (of bounded size) of the implementations is corrupted, the combination is still secure. Such robust combiners not only aim to mitigate accidental failures, but can even deal with completely adversarial implementations. Previous work exists on finding a formal definition and gives various easy combiners like cascade or copying, giving an overview of easy constructions [56]. It has been shown that some robust combiners are impossible to create in a black-box way, i.e. by using the different implementations

only via their input-output interface and not via their code, for example for oblivious transfer [54, 89].

Robust combiners have also been constructed for more complex hardware components like firewalls [3]. Similar to our approach, a trusted interface that sends the inputs to the individual implementations and performs a check on the outputs (for example a comparison or a majority vote), is used.

4.1.2. Contribution and Outline

In this work, we propose, model and implement an approach for increasing the security of software execution on untrusted general-purpose systems. In particular, we consider the setting of two untrusted systems, possibly from different vendors and running different operating systems, that execute QEMU Linux VMs in a record-replay manner. To distribute input to these systems as well as to compare their outputs, we use a trusted input-output interface (see fig. 4.1). We prove that if one host correctly executes the VM and if the input-output interface is trusted, then the only adversarial influence to the execution constitutes of aborting and controlling non-deterministic behavior. In particular, either the output is correct (relative to the allowed adversarial influence) or no output is given at all.

Our approach is, to the best of our knowledge, the first to take special care to ensure that the used randomness is cryptographically secure, making it suitable for a setting where security and privacy are important. We demonstrate the practicality by realizing a system based on QEMU and measuring its performance.

In more detail, we provide a description of our approach, a threat model as well as a formal model and proof of security in section 4.2, discuss possible applications in section 4.3 and give an overview of our implementation as well as benchmark results in section 4.4.

We stress our goal is not to protect the software inside the VM from being bug-free, and thereby from malware attacks or hacks. Instead, our focus is ensuring its correct execution and implied security and privacy guarantees for appropriate programs when at least one VM host and the input-output interface are honest.

Our approach is suitable for a setting where a very high level of security is needed, at the expense of performance or system complexity. In particular, CoRReCt may be helpful when trusted general purpose systems cannot be easily obtained or when hardware trojans or supply chain attacks are a valid threat.

4.2. The CoRReCt Approach

We start the presentation of CoRReCt with an informal description, continue with a threat model and finish with a formal model. Within this model, we state an ideal functionality that captures the security guarantees provided by CoRReCt. Subsequently, we cast our approach as a protocol within our model and prove that it realizes the ideal functionality.

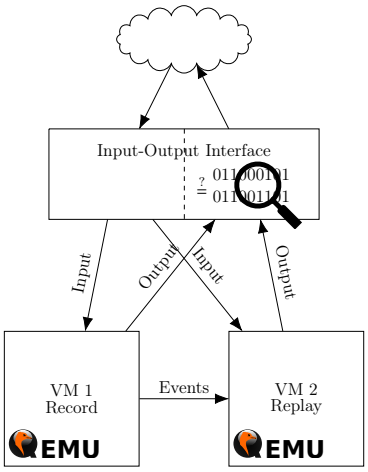

Figure 4.1.: Architecture of a system protecting a computation using record-replay on x86-64 machine level. Events are QEMU-internal events, inputs and outputs are ethernet packages

4.2.1. Description

Following the setting illustrated in fig. 4.1, we consider an execution with three machines. One machine, called IO, is tasked with distributing inputs to two other machines denoted by P_1 and P_2 that execute the same program. P_1 is called the recorder. It executes a Turing machine μ on inputs received by IO and records all non-deterministic events. Conversely, P_2 is called the replayer. It receives inputs from IO, as well as a trace of recorded non-deterministic events for each activation from P_1. Using the input and the trace, P_2 also executes μ. Both P_1 and P_2 provide output to IO. To determine the contents of μ's random tape, P_1 and P_2 perform an appropriate protocol. If the outputs of P_1 and P_2 match, IO allows the output. Otherwise, it halts without accepting further inputs or outputs. We assume that P_1, P_2 and IO may directly communicate with each other, without any adversarial influence beyond the corruption of parties.

4.2.2. Threat Model

We continue with an introduction of the threat model underlying CoRReCt. Looking ahead, we propose a formal model capturing this informal threat model in section 4.2.4.

CoRReCt considers the execution of software on commodity hardware, say on the Intel x86-64 architecture in the presence of an adversary. In line with the standard adversarial model in cryptography, CoRReCt considers intelligent adversaries. In particular, the only restriction we impose on adversaries is that their computational power is polynomially bounded. We distinguish between the system (i.e. the hardware including its firmware), and the application (i.e. software deployed by the user, including the operating system, system libraries, ...). CoRReCt can also be applied to executions where this distinction is made on a different abstraction level ("System" up to OS, "Application" all userspace libraries; "System" up to Java Virtual machine, "Application" all Java Code), but for the rest of this work we will focus on the x86-64 machine as dividing line.

Guarantees of an Honest Execution. First, we consider the expected security guarantees of software execution on a single machine in the setting outlined above.

Even if software and hardware are uncorrupted, the execution of software is subject to non-determinism at various levels, for example due to external input and output, network messages, jitter or scheduling decisions of the operation system. In contrast to randomness, which is usually assumed to be of high entropy, no guarantees with respect to the behavior or distribution of non-determinism is made. Consequently, even "unexpected" non-deterministic behavior should not negatively affect the execution. (Of course, in the absence of a formal specification, none of this is well-defined.) The fact that software and hardware are assumed to be uncorrupted does not rule out that they contain bugs that lead to undesirable behavior from the perspective of the user, for example incorrect computation results. Also, the software and hardware may be subject to vulnerabilities that can be exploited by adversaries, resulting in corrupted software or hardware that is discussed below.

Guarantees of an Execution on a Single Corrupted System. Clearly, if hardware or software are corrupted, little security guarantees remain. If only the software is corrupted, for example due to the exploitation of vulnerabilities, the adversary may learn all secrets and perform arbitrary changes to the execution. Still, hardware-based security may remain, for example the privacy of keys stored in a trusted platform module (TPM). Also, if hardware and firmware are uncorrupted, the system may be returned to an honest state by reinstalling the software.

Conversely, if the hardware is corrupted, the situation is even worse, because corrupted hardware may deviate arbitrarily from the software execution, and security supposedly provided by the hardware are lost as well. In particular, corrupted hardware could leak stored cryptographic keys or unfaithfully execute hardware-based cryptographic protocols, for example for remote attestation. Also, recovering from corrupted hardware may be much harder. Even if only the firmware is corrupted, physical access with special tools may be needed.

Adversarial Model of CoRReCt. In contrast to the previous two paragraphs, which considered executions on a single machine only, CoRReCt considers a distributed execution on machines P_1 and P_2 whose results are compared by an input-output interface IO. In more detail, we assume that the machines P_1 and P_2 are commodity hardware running commodity software. We do not assume that there exists a formal specification of correctness, let alone a proof of correctness or security for the software or for P_1 and P_2. Also, we again assume that P_1 and P_2 are connected to the Internet (via IO). For the sake of simplicity, we assume that all external entities are adversarially controlled.

In this setting, CoRReCt aims to protect an honest user against the consequences of corrupted hardware. Other goals such as protecting against the consequences of corrupted software are out of scope as discussed below.

Adversaries may corrupt P_1 or P_2 in a number of ways. For example, hardware or firmware of P_1 or P_2 may already be corrupted when deployed by the user, for example due to a supply chain attack. Corruption may also occur after P_1 and P_2 are deployed, for example because an adversary was able to remotely exploit a vulnerability or get physical access to a machine. CoRReCt aims to protect against the consequences of corrupted hard- and firmware as long as only one of P_1 and P_2 is corrupted and IO is honest. While there CoRReCt cannot prevent corruption, we believe that this corruption model is justified by the fact that a) the honest user is able to prevent physically tampering of the hardware once deployed, b) by (anonymously) sourcing commodity hardware from different sources, the risk of supply chain attacks can be

reduced and, finally, c) due to the use of heterogenous systems, the risk that both manufacturers produce corrupted systems is reduced.[3]

The focus of CoRReCt on hard- and firmware corruptions is due to the fact that they may be very hard to prevent, detect or fix. This is because hardware and firmware often are closed-source, making it impossible to judge whether it is uncorrupted or not. Even reliably determining whether a certain firmware is used is often very hard in practice. Thus, it is desirable to at least mitigate their negative consequences as much as possible.

CoRReCt cannot and does not want to protect against the consequences of corruption of the software. Indeed, if the software is corrupted, few if any security guarantees may be left. However, we believe that the problem is more easily solved. To this end, tools like memory-safe programming languages or even formal verification are available. However, if hard- or firmware are corrupted, even software that is formally verified for correctness and the absence of vulnerabilities may not provide any benefits, because the hardware can deviate from the software's execution in an arbitrary way.

Adversarial Influence in CoRReCt. In a setting where the assumptions about the adversarial model hold, the adversary's influence is limited as follows:

- It can arbitrarily interact with the software, possibly exploiting vulnerabilities.
- It can arbitrarily control the hardware of the corrupted system.
- It can arbitrarily control the timing of the execution if either machine is corrupted.
- It can force the stop of the execution either directly on the corrupted system or by presenting an output of the corrupted system that does not match the output of the honest system. In this case, IO will abort.
- Finally, it can arbitrarily control the non-determinism exposed to the software if the recording machine P_1 is corrupted.

While the first four points are clear and natural, the last point merits further discussion.

Impact of Non-Determinism. To illustrate the (perhaps surprising) consequences of (adversarially-controlled) non-determinism in the execution, consider the following example. Let p be a program that does the following: Repeatedly sample a bit b using sources of non-determinism. If $b = 0$, read k random bits, for example through a source of randomness provided by the operating system, but do not generate output. If $b = 1$, also read k random bits and output them. This program illustrates the influence of non-determinism, e.g. on a system with multiple processes and a preemptive scheduler between them where one process is the application we are interested in (when $b = 1$), and the other is some irrelevant processes, requiring randomness as well (when $b = 0$).

Suppose that the adversary controls the recording machine of CoRReCt. In this case, it is aware of the system's randomness and may fully choose the bit b each time. Let us assume the adversary to choose $b = P(r)$ as a general predicate of the read bits $r \in \{0,1\}^k$. Let $p := \Pr[P(r) = 1]$, for

[3] While not formally considered, CoRReCt may even offer security guarantees if both P_1 and P_2 are corrupted, but do not cooperate. This may, for example, be the case if the corruptions originated from state-level attackers of hostile states.

$r \in \{0,1\}^k$ chosen uniformly at random. In this case the expected number of non-deterministic choices before an output is made is: $p \cdot 1 + p(1-p) \cdot 2 + p(1-p)^2 \cdot 3 + \ldots = \frac{1}{p}$.

The value of the chosen r is uniformly distributed among all $n := p \cdot 2^k$ possible values with $P(r) = 1$. Thus, the entropy for an adversary knowing the sampling procedure is $\sum_{i=0}^{n} -\frac{1}{n} \log(\frac{1}{n}) = \log(n) = k + \log p$ and an attacker spending additional time t can reduce the entropy in the randomness by $\log(t)$.

This relation illustrates how important it is to obtain randomness from a shared generator in large enough blocks k, as this sets the entropy baseline that the attackers runtime is weighted against. Also, it illustrates the perhaps surprising influence of adversarially-chosen non-determinism of an otherwise honest execution. An obvious consequence of such a setting is that programs should not assume that the non-determinism is "benign", but should be able to deal with the worst case.

For an application that requires randomness to obtain e.g. a cryptographic key inside CoRReCt, it is vital, that it gets its output from the system PRNG in one block instead of several smaller blocks. Tinfoil Chat, one of the applications we will present later in section 4.3.2, obtains its randomness in one block[4].

4.2.3. Corruptions and Non-Determinism

Consistent with the problem we intend to address, namely the lack of trust in today's systems, and in line with our threat model (section 4.2.2), we consider different levels of adversarial influence on the execution of μ: Even if all entities are honest, we consider non-deterministic aspects of the execution of μ to be under adversarial control. To this end, the adversary is allowed, for every machine μ' that is part of the execution, to specify a special (efficient) Turing machine μ'_N. μ'_N has access to all state of μ' and may, depending on this state, determine non-deterministic parts of the execution. Intuitively, this captures the freedom a machine implementation has when choosing which non-deterministic option to take. This is consistent with our expectation that "useful" programs need to be able to deal with arbitrary non-determinism that may influence their execution. For a program not affected by non-determinism, this trivially guarantees a correct execution (if the machine executing the program is honest). Still, a program might often employ actions that are non-deterministic. Examples include user input or output or network communication. Then, the program must be able to deal with the resulting non-determinism. In particular, this mechanism allows to capture leakage of secrets through non-deterministic aspects of an execution if μ (i.e. the program) and μ_N are designed appropriately (see section 4.2.10).

If P_1 is corrupted, we assume that the adversary learns the inputs of P_1, the code of μ as well as the random tape of μ. Also, in case of corruption, P_1 is fully controlled by the adversary and may deviate from its specified behavior arbitrarily. Additionally, the adversary may adaptively and arbitrarily determine non-deterministic parts of the execution. In contrast to the influence on non-corrupted machines, an adversary can make a corrupted machine deviate arbitrarily from the machine specification and choose its output.

[4] https://github.com/maqp/tfc/blob/07a819b3a8e1ce98bfc292b0ee8a76cb713e9645/src/common/crypto.py#L709

If P_1 is corrupted but P_2 is honest, we can prove that the only possibility for the adversary to influence the execution of μ (except for aborts) is to adaptively determine the non-deterministic parts of its execution: If a corrupted P_1 deviates from its protocol in a way that influences the (presumptive) outputs of μ, then the honest party P_2 will produce a different output that is detected by IO, leading to a halt of the execution.

Conversely, if P_2 is corrupted but P_1 is honest, the guarantees are very similar to the previous case, with the exception that the non-deterministic parts of the execution are governed by μ_N of P_1 and not adaptively by the adversary.

We assume that IO is honest at all times. This is motivated by the fact that the functionality of IO is very simple and may be implemented in a trusted and verifiable way, for example as a fixed-function circuit.

4.2.4. Model

In order to formally capture the security provided by our approach, a model is needed. To this end, we use a simplified variant of the Universal Composability (UC) framework [23].

The basic machine model of UC is a so-called interactive Turing machine. We retain this machine model, but augment it to model non-determinism of program execution encountered in the real world. To this end, we introduce an additional non-determinism tape that can be accessed read-only. We call the resulting machine model interactive Turing machines with non-deterministic events, formally defined as follows:

Definition 16 (Interactive Turing Machine with Non-Deterministic Events) An interactive Turing machine with non-deterministic events is defined like an interactive Turing machine [23] with an additional read-only non-determinism tape. This tape is initiated with uniform randomness.

We stress that, as discussed above, the non-determinism tape does not model non-deterministic Turing machines. In particular, the complexity class we consider remains $\mathcal{BPP}$. The tape's purpose is to provide an explicit mechanism for possibly adversarially controlled non-determinism.

For a (real) program under analysis, all non-determinism that is to be considered during the execution must be modeled through behavior depending on the content of the non-determinism tape.

We define the non-determinism trace of an activation of a Turing machine as the sequence of symbols read from the non-determinism tape.

Definition 17 (Non-Determinism Trace) Let c_1 be the configuration of an (interactive) probabilistic Turing machine with a non-determinism tape. Upon activation on some input that ends in configuration c_2, the non-determinism trace t is the sequence of symbols read from the non-determinism tape.

We consider the following entities: An interactive distinguisher $\mathcal{Z}$, called environment, that interacts with an adversary $\mathcal{A}$ as well a protocol π comprised of "main parties" P_1, P_2 and IO. Looking ahead, the task of $\mathcal{Z}$ will be to distinguish between an execution of a protocol π_1 and an adversary $\mathcal{A}$ and a protocol π_2 and an adversary $\mathcal{S}$ called the simulator.

Informally, the simulator's task is to simulate the execution of π_1 and $\mathcal{A}$ for $\mathcal{Z}$, even if π_2 is actually executed. Usually, π_2 will be a protocol that is secure by definition. If the indistinguishability of these two executions can be shown, then all properties guaranteed by π_2 also hold in an execution of π_1.

4.2.5. Execution Experiment

We now define the execution experiment. This execution captures the setting described in section 4.2.1 and is closely modeled after the UC execution[5].

Definition 18 (Execution Experiment) Let $\mathcal{Z}$ and $\mathcal{A}$ be polynomial-time interactive Turing machines. Let P_1, P_2 and IO be polynomial-time interactive Turing machines with non-deterministic events. Let π be a protocol with main parties P_1, P_2 and IO.

We consider the following execution with machines $\mathcal{Z}$, $\mathcal{A}$ and a protocol π.

1. On input $(1^\kappa, z)$ for the execution, a machine called environment is invoked with the code of $\mathcal{Z}$ and input $(1^\kappa, z)$.
2. On its first activation, the environment invokes the adversary, which is given the code of $\mathcal{A}$ and an input chosen by the environment.
3. On its first activation, the adversary may corrupt an arbitrary subset of parties in the set $\{P_1, P_2\}$. Corrupted parties are under full adversarial control and may deviate from the protocol arbitrarily. Let C denote the set of corrupted parties. For each honest party P in the set $\{P_1, P_2, IO\} \setminus C$, the adversary may specify the description of a probabilistic polynomial-time Turing machine μ_P that adaptively provides the values on the non-determinism tape of P relative to the code and contents of all other tapes of P. Then, it activates the environment again.
4. The environment may provide input to IO, whose code is specified in π and receive output from IO.
5. IO, P_1 and P_2 act as specified in the protocol π, which may entail the creation of further sub-parties that behave as specified in the protocol.
6. The environment and the adversary may communicate with each other at any point. Moreover, the adversary may participate in the execution on behalf of corrupted parties.
7. The environment eventually outputs a bit b and halts.
8. Unless noted otherwise, all communication between honest entities is ideally secure and immediate, i.e. oblivious for the adversary.

[5] The changes to incorporate non-determinism would require to re-prove all properties of UC security. As we do not consider composable security, we consider this to be out of scope for this contribution. Instead, we craft a custom model based on UC without establishing possible properties.

4.2.6. Ideal Functionality

With the execution experiment at hand, we define an ideal functionality $\mathcal{F}$ that captures the security guarantees provided by our approach. By definition, $\mathcal{F}$ is incorruptible and parameterized with a program μ in the form of the code of a probabilistic polynomial-time Turing machine. Also, $\mathcal{F}$ interacts with parties IO, P_1 and P_2 as well as an adversary.

Initially, $\mathcal{F}$ samples a new random tape for μ. Then, μ is executed inside $\mathcal{F}$ based on inputs provided by IO to $\mathcal{F}$. Outputs of μ are done through IO. Depending on which parties are honest or corrupted, the adversary is able to influence the execution or learn information in several ways:

- If all entities are honest, the adversary may provide a machine μ_N that provides the content of the non-determinism tape for μ, depending on μ's current state. However, the adversary neither learns inputs or outputs, nor the randomness of μ through $\mathcal{F}$.
- If only either P_1 or P_2 are corrupted, the adversary learns the inputs and randomness of μ. If P_1 is corrupted, the adversary may adaptively choose the contents of μ's non-determinism tape.
- If both P_1 and P_2 are corrupted, there are essentially no guarantees.

We assume that IO is always honest.

Definition 19 (Ideal Functionality $\mathcal{F}$ for CoRReCt) Parameterized by a TM μ with non-determinism tape with alphabet Γ, parties P_1, P_2, IO and a security parameter κ.
Initially, set $state = \bot$, initialize an empty list $\mathcal{L}$ and sample a fresh random tape for μ.
Corruption and Non-Determinism.

- If P_1 and P_2 are honest, let the adversary provide the description of a probabilistic polynomial-time Turing machine μ_N. Whenever μ reads from its non-determinism tape, execute μ_N on the current configuration of μ, resulting in output $\gamma \in \Gamma$. Report γ to μ as the current symbol on its non-determinism tape.
- If at least one P_i is corrupted (for $i \in \{1, 2\}$), send the contents of the random tape and the description of μ to the adversary. Also, whenever μ attempts to read a symbol from its non-determinism tape and P_1 is corrupted, send $(\texttt{query})$ to the adversary and receive $(\texttt{answer}, \gamma)$ with $\gamma \in \Gamma$. Provide μ with γ as the symbol read from the non-determinism tape.
- If both P_1 and P_2 are corrupted, let the adversary determine all outputs.
- On message $(\texttt{query} - \texttt{io})$ from the adversary:
  - If at least one P_i is corrupted, send $(\texttt{query} - \texttt{io}, \mathcal{L})$ to the adversary.
  - Otherwise, send $(\texttt{query} - \texttt{io}, \bot)$ to the adversary.

Execution.

- On input $(\texttt{input}, m)$ for IO, add m to $\mathcal{L}$ and execute μ on input $(1^\kappa, state, m)$. Let m' be the output of μ and let $state'$ be the state after the execution. Save $state = state'$, add m' to $\mathcal{L}$ and generate a private
  - output $(\texttt{output}, m')$ if P_1 and P_2 are honest resp.

– delayed output $(\texttt{output}, m')$ if P_1 or P_2 is corrupted

for IO. Accept further inputs.

As a consequence, the security guarantees provided by $\mathcal{F}$ are only as good as the security guarantees of μ are relative to the considered model of execution. In particular, they depend on the influence non-determinism has on the output(s) of μ. In particular, μ_N can possibly be used to exfiltrate secrets, even if P_1 and P_2 are honest. For a given program to be analyzed, determining the right amount of non-deterministic influence in μ and the behavior of μ_N is up to the analyst.

In the following, we define the protocol $\pi_{\mathcal{F}}$ that wraps an execution of $\mathcal{F}$, similar to an ideal protocol in the UC framework [23]:

Construction 6 (The protocol $\pi_{\mathcal{F}}$) Let $\pi_{\mathcal{F}}$ be the following protocol for the parties P_1, P_2 and IO, parameterized with a Turing machine μ. Let $\mathcal{F}$ be the ideal functionality of definition 19.

- P_1 and P_2: Ignore all inputs and all outputs. In case of corruption, inform $\mathcal{F}$ of the corruption.
- IO:
  - On input $(\texttt{input}, m)$, send input $(\texttt{input}, m)$ to $\mathcal{F}$.
  - On output $(\texttt{ouptut}, m')$ from $\mathcal{F}$, give output $(\texttt{output}, m')$.

As P_1, P_2 and IO are deterministic by definition in this idealized execution, we do not need to consider their non-determinism tapes.

4.2.7. Protocol

We now state a protocol that captures the approach presented in section 4.2.1 in our model. Informally, π works as follows: On the first input (of IO), P_1 and P_2 call the ideal functionality for coin-tossing $\mathcal{F}_{ct}$ (definition 21) to obtain a uniformly random string, which is used as the random tape for the execution of the TM μ. P_1 acts as recording machine executing μ and evaluating all non-deterministic events by reading from its non-determinism tape. These non-determinism traces are subsequently provided to P_2 in order to be able to replay the execution. Both P_1 and P_2 provide their outputs to IO. If both outputs match, IO generates an output. Conversely, IO also distributes inputs to P_1 and P_2. For honest IO, P_1 and P_2, we assume that they directly communicate without adversarial influence.

Construction 7 (The CoRReCt Protocol π) π is parameterized with a probabilistic polynomial-time TM μ. The main parties of π are P_1, P_2 and IO.

Protocol of IO:

1. Initiate $o_1, o_2 = \perp$.
2. On input $(\texttt{input}, m)$:
   - If this is the first input, send $(\texttt{coin-toss})$ to P_1. After having received $(\texttt{coin-toss})$ from P_1, and for every subsequent input, continue as follows:
   - Send $(\texttt{input}, m)$ to P_1 and receive $(\texttt{output}, m_1)$ from P_1. Activate P_1 again.
   - Send $(\texttt{input}, m)$ to P_2 and receive $(\texttt{output}, m_2)$ from P_2.
   - After having received $(\texttt{output}, m_i)$ from both P_1 and P_2: If $o_i = \perp$, set $o_i = m_i$. If $o_1 = o_2 \neq \perp$, output $(\texttt{output}, o_1)$ and set $o_1 = o_2 = \perp$. If $o_1 \neq o_2$ and $o_1 \neq \perp$ and $o_2 \neq \perp$, halt.

Protocol of P_1:

1. On message $(\texttt{coin-toss})$ from IO:
   a) Send $(\texttt{coin-toss})$ to $\mathcal{F}_{ct}$.
   b) Receive $(\texttt{coin-toss}, r)$ from $\mathcal{F}_{ct}$. Store r and use r as randomness for the execution of μ.
2. On message $(\texttt{input}, m)$ from IO:
   a) Set $t = \varepsilon$.
   b) Continue the execution of μ on input m, recording a non-determinism trace t.
   c) Store the current state of μ and send $(\texttt{output}, m')$ to IO, where m' is the output of μ.
   d) On the next activation, send $(\texttt{trace}, t)$ to P_2.

Protocol of P_2:

1. On message $(\texttt{coin-toss})$ from IO:
   a) Send $(\texttt{coin-toss})$ to $\mathcal{F}_{ct}$.
   b) Receive $(\texttt{coin-toss}, r)$ from $\mathcal{F}_{ct}$. Store r and use r as randomness for the execution of μ.
2. On message $(\texttt{trace}, t)$ from P_1, store t and activate IO.
3. On message $(\texttt{input}, m)$ from IO:
   a) Continue the execution of μ on input m and trace t for the non-determinism tape. If there is no stored trace t, halt.
   b) Delete t, store the current state and send $(\texttt{output}, m')$ to IO, where m' is the output of μ.

We now state our main theorem, namely that our protocol π realizes the ideal functionality $\mathcal{F}$ for CoRReCt. Thus, π is "just as secure" as the ideal functionality $\mathcal{F}$, meaning that all security guarantees of $\mathcal{F}$ also hold in an execution with π.

Theorem 6 For every PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that for every PPT environment $\mathcal{Z}$, the output of $\mathcal{Z}$ in the execution of π with $\mathcal{A}$ and the execution of $\pi_{\mathcal{F}}$ and $\mathcal{S}$ are identically distributed.

For the proof, see section 4.2.8.

4.2.8. Proof

We now prove theorem 6.

Proof 29 To this end, we state a simulator. To make the presentation easier, we consider a simulator for the so-called dummy adversary [23], which is the adversary that reports all messages it sees to the environment and delivers all messages it receives from the environment. Intuitively, this adversary is the hardest to simulate. For UC security, one can show that the dummy adversary is complete, i.e. if there exists a simulator for the dummy adversary, then there also exists a simulator for every other PPT adversary [23]. It is easy to see that this also holds for our model.

Informally, the task of the simulator is to simulate an execution of π and $\mathcal{A}$ for the environment, while actually $\pi_{\mathcal{F}}$ is executed. If the simulator succeeds with this task, then all properties of $\pi_{\mathcal{F}}$ (and $\mathcal{F}$) also hold in π.

Definition 20 (Simulator) We distinguish between the following cases:

All Parties Honest Let μ_N be the machine the environment provides to fill the non-determinism tape of P_1. Send the description of μ_N to $\mathcal{F}$.

P_1 Corrupted but P_2 Honest:

1. Obtain r from $\mathcal{F}$, report $(\texttt{coin-toss}, r)$ as output of $\mathcal{F}_{ct}$ for P_1.
2. Internally, emulate an instance I_2 of μ, using random tape r.
3. On each activation of P_1, send $(\texttt{leak-io})$ to $\mathcal{F}$ and get the current input m.
4. When necessary, query non-deterministic events from the environment and provide them to $\mathcal{F}$.
5. If the environment instructs the adversary to send a different $(\texttt{trace}, t')$ message to P_2 (relative to the trace provided to $\mathcal{F}$), execute I_2 on its current state, input m and the trace t'. If the environment does not instruct the adversary to send a different trace, execute the instance I_2 using the non-deterministic events provided to $\mathcal{F}$.
6. If the environment instructs the adversary to send a different `output` message from P_1 to IO (relative to the output of I_2), do not allow the corresponding output of $\mathcal{F}$.

P_2 Corrupted but P_1 Honest:

1. Let μ_N be the machine the environment provides to fill the non-determinism tape of P_1. Send the description of μ_N to $\mathcal{F}$.
2. Obtain r from $\mathcal{F}$, report $(\texttt{coin-toss}, r)$ as output of $\mathcal{F}_{ct}$ for P_2.
3. Internally, emulate an instance I_1 of μ, using random tape r and μ_N for the non-determinism tape. Send the traces of the execution to P_2 on behalf of P_1 when P_1 would do so in the protocol.
4. On each activation of P_2, send $(\texttt{leak-io})$ to $\mathcal{F}$ and get the current input m.
5. Execute instance I_1 of μ on its current state with input m, using the non-deterministic events provided by μ_N and randomness r.
6. If the environment instructs the adversary to send a different `output` message of P_2 to IO (relative to the output of I_1), do not allow the corresponding output of $\mathcal{F}$.

Both P_1 and P_2 Corrupted:

1. Whenever IO would make an output in π, make the corresponding output through $\mathcal{F}$.

By the definition of $\mathcal{S}$, it follows that the two executions are identically distributed and the claim follows.

4.2.9. Ideal Functionality $\mathcal{F}_{ct}$

The ideal functionality for coin-toss $\mathcal{F}_{ct}$ is defined as follows:

Definition 21 (Ideal Functionality $\mathcal{F}_{ct}$) Parameterized with parties P_1, P_2, a length parameter l and an adversary.

1. On first input $(\texttt{coin-toss})$ from a party $P \in \{P_1, P_2\}$, sample $r \leftarrow \{0,1\}^l$ uniformly at random. Output $\texttt{coin-toss}, r)$ to P.
2. On subsequent input $(\texttt{coin-toss})$ from the other party $P' \in \{P_1, P_2\}$, output $(\texttt{coin-toss}, r)$ to P'.
3. Ignore all subsequent inputs.

4.2.10. Information Leakage Example

Consider a program that evaluates a (deterministic) and efficient function $F(x)$ and in addition samples non-determinism d and outputs $(F(x), d)$. The adversary might now specify μ_N (see section 4.2.1) to compute an approximation for the number of steps of $F(x)$ during execution, such that this information is embedded in d. The result now consists of $F(x)$ and d, effectively allowing to capturing many possible side-channel attack scenarios.

4.3. Applications

We discuss a series of applications of the CoRReCt concept that vary in their security requirements.

4.3.1. Secure Multi-Party Computations

In a setting where the strong security guarantees of secure multi-party computations (MPC) are desirable, using CoRReCt to participate in the protocol offers an interesting trade-off: On the one hand, if at least one system performing the party's computation is honest and an output is returned, CoRReCt guarantees that this output is the actual output of the computation. On the other hand, CoRReCt may negatively impact the privacy: In a k-round protocol, $\log_2(k)$ bits can be leaked simply by one of the systems aborting at a point of its choice.

Additionally, a malicious recording system may use timing side-channels to leak information. If the same (malicious) system would have been used to participate in the computation when not using CoRReCt, there would not have been any security guarantees. In particular, it could use different inputs for the computation, present different outputs and leak all secrets to the adversary. Thus, under this corruption, the timing side-channel that still exists with CoRReCt may not constitute a loss of security. However, if the recording system were honest, a malicious replaying system could still leak information by aborting, which constitutes a loss in security. In order to protect from timing side-channels of the replaying system, outputs must be time-synchronized: If the recording system wants to output a value x at time t, this output may only be allowed at time $t + T$ if the replaying system also outputs x at time t' in the interval $(t, t + T)$. This reduces the possible influence of the replaying system to aborts at the expense of additional latency governed by the parameter T.

For certain cases, better guarantees may be obtained: If the party has no private input (e.g. because it is the receiver of a commitment) and its output is computationally hidden until the last message is received, the leakage can be reduced to a single bit. This even works in case of a corrupted recording system if a timeout for returning the result after receiving the last message is enforced.

4.3.2. Tinfoil Chat

Tinfoil chat [87] aims to provide a high-security solution for secure communication. To this end, four dedicated systems are used: A source computer which receives outgoing messages from the user and is connected to a data diode such that it can only send messages, but not receive them. In particular, the source computer is not directly connected to the network. Conversely, a destination computer handles incoming messages. It is also attached to a data diode, but in the opposite direction such that it can only receive messages, but not send them. All network communication is handled through a networked computer.

The rationale behind this architecture is as follows: If the source computer is initially honest, it cannot be compromised via the network. Thus, plaintext messages are protected from hacks. While the destination computer can receive messages and is thus susceptible to hacks, the data diode prevents exfiltration of keys used for decryption as well as decrypted plaintexts. Note that

integrity is not guaranteed in case of corruption. As the networked computer does not handle secrets, it may be corrupted without affecting security.

Taking a closer look, we observe the following: The source computer repeatedly generates cryptographic key material. To this end, it relies on the Linux getrandom syscall, which is fed from several sources of non-determinism, which is extensively discussed in the Tinfoil chat code. Thus, Tinfoil chat makes an implicit assumption about this non-determinism to be of sufficient quality. If the quality were insufficient, security might be affected. If the source computer were malicious, it could modify the execution to leak secrets, e.g. through the selection of randomness of ciphertexts such that the least significant bit of the ciphertext leaks information about key material or plaintexts.

While these problems are out of scope for Tinfoil chat (which explicitly states that it does not protect from "interdiction of hardware"), they are in the scope of CoRReCt. In particular, if CoRReCt is used for the source computer, it can ensure that a) the adversarial influence with respect to the choice of the randomness is limited and b) that malicious hardware or a malicious host operating system could not manipulate the execution. Given that Tinfoil chat also protects (to some extent) from timing side channels through network messages, the level of protection resulting from the combination of both approaches promises to be very strong. We also note that the computations performed by Tinfoil chat are not very computation-heavy. Looking ahead at the measured overhead of our approach, we consider the degradation of performance acceptable, in particular if one deems it necessary to use a system like Tinfoil chat in the first place.

The data diode needs to be extended to also compare the output of the two executions, and to copy the keyboard input to for both source computers. For a concrete implementation a RP2040 microprocessor could be used to compare the contents of two UART streams (which are used by the proposed DIY data diodes), and if equal, forward them to the data diode. Using its USB interface a USB keyboard can be read out and fed to both source systems equally via the USB-UART converters. As input from the keyboard will be be echoed on both source computers, a corrupt keyboard cannot change the input unnoticed. The user would need to keep an eye on both source computers' displays before relying on the messages displayed there. Care needs to be taken that the code running the comparison cannot be overridden e.g. from the two source computers.

To test if the performance of CoRReCt is sufficient to run Tinfoil chat, we started a test installation of all Tinfoil chat components within CoRReCt. For inputs into CoRReCt we observed an input delay of approximately 200ms on average, being noticable, but not hindering usage. Sending messages with Tinfoil chat via TOR already incurs some latency so that additional latency introduced by CoRReCt was not noticable.

4.3.3. Access Control

Evaluating access control policies can be complex and a high value target for an attacker. When the output of this application is deterministic, CoRReCt can be used to ensure the integrity of the resulting access decision. However, a third party may need some way to authenticate the output of this computation. Using a signing key, as in the general server application, requires the elimination of non-determinism both on protocol level (e.g. TLS or something simpler) as well as on application level to prevent unintended leakage of cryptographic secrets.

	Repr. Builds	SGX	SEV/TDX	CoRReCt
Performance	✓	✓	✓	TCG overhead single-threaded
Auditability	✓	×	×	✓
Security not based on HW trust	✓	×	×	1-of-2 + simple comparator
Applicable to generic build process	×	?	✓	✓

Table 4.1.: High-level comparison of the advantages and disadvantages of the different approaches for software building.

4.3.4. Software Building

Systems used to compile software are a valuable target for supply-chain attacks. Reproducible builds[6] make software build processes deterministic by removing any influence non-determinism might have on the artifacts, greatly limiting an adversary's influence. When the build process cannot be adjusted, there are scientific approaches to reduce non-determinism to some extent [84], but there are still many special cases that are not handled. In case of proprietary software, the build process can be very complex, as it might involve the usage of closed-source tools, automatic dependency downloads and other software that makes the process non-deterministic without the possibility to troubleshoot difficult. In this case, CoRReCt can help to verify the remaining non-determinism in a build process. What is more, the event stream and build system image can be kept on record to allow to re-verify the built artifacts later for audit purposes.

Table 4.1 shows a high-level comparison of different approaches to securing software building. SGX and SEV as commercially available enclaves aim to ensure correctness and integrity by relying on a vendor created key. AMD SEV and Intel TDX, as concepts that target VMs, are more general and heavyweight, while SGX focuses on isolating individual applications. CoRReCt, in line with AMD SEV and Intel TDX, allows ensuring the correctness of a whole VM execution and is therefore the most generally applicable.

4.4. Realization of CoRReCt with QEMU Linux VMs

In this section, we present a realization of our approach, with QEMU as a machine emulator at its core. In more detail, we consider the execution of Linux VMs for the x86-64 architecture with remote network access, motivated by the "Tinfoil Chat" and "software building" use-cases presented in section 4.3. To this end, we modified both QEMU as well as the Linux kernel as outlined below.

We rely on QEMU's record-replay for deterministic replay by first recording non-deterministic events during the execution of a virtual machine. These events are written to a file and used as input during replay. Technically, an event consists of an ID to determine what kind of event was recorded, together with associated data. Some examples are network events, clock events and interrupt events. To be able to replay, a recording QEMU VM additionally stores the number of executed instructions in the event log. During replay, this number is used to inject recorded events at the right moment.

[6] https://reproducible-builds.org

By default, record-replay first records an execution and replays it after the recording has finished. For CoRReCt, we modified QEMU to enable the "pseudo-parallel" execution of recording and replaying. To this end, two separate QEMU instances are created at the same time—one for recording and one for replaying. When the recorder starts the execution, it sends all events directly to the replayer. The replayer immediately replays the recording by using the event stream as input. We call this pseudo-parallel because of the delay for transmitting events.

Providing Randomness. To provide randomness that is suitable for cryptographic applications and not generated by harvesting non-determinism, it was necessary to implement changes to a) Exchange a seed for initializing a random number generator (RNG), b) pass a stream of random bytes from this RNG to QEMU, and c) modify the Linux kernel's RNG to only use the provided randomness.

Passing randomness to QEMU is possible by specifying a file to read from. This file's contents are forwarded to the VM and presented as a hardware RNG, accessible from userspace through /dev/hwrng. Linux' RNG uses this device file by default as source for its entropy pool. On top of that, it uses additional sources of randomness that are not suitable for our purpose because they rely on non-determinism, e.g. interrupt timings. As the values of theses sources can be set unilaterally by the recording VM, we modified the Linux kernel to only use /dev/hwrng as source of randomness. The generation of a suitable random stream happens outside of QEMU. This is done by exchanging a seed for a RNG between the recorder and replayer based on the coin tossing protocol of Blum [15]. After the exchange is finished, the seed is used to initialize a PRNG.

Handling Network Traffic. To handle network traffic, there exists a third system that is connected to both QEMU instances and is called input-output interface. Those connections are established by using QEMU's socket network backend. The interface's first task is to send incoming packets from the network to both virtual machines. Its second task is to receive outgoing packets from both virtual machines, compare them and, if they match, forward them to the network. If outgoing network packets do not match, execution is stopped. To enable the usage of the interface, QEMU was modified such that network packets can be injected and sent during replay. The default settings for replay mode discards outgoing packets and reads incoming packets from the event log file. Enabling transmission of outgoing packets required changes to QEMU's network filter for record-replay such that outgoing packets are not discarded. Packet injection is achieved by storing network packets sent by the interface in a separate list inside of QEMU. This list is then used whenever a packet is read from the event log to replace the network packet from the log with the one injected by the interface. An additional property of this system is that network traffic is inherently ordered because of deterministic replay.

Benchmarks. tables 4.2 and 4.3 show benchmarking results for three different scenarios on two machines with Intel Xeon D-1718T CPUs. First, QEMU is run with KVM, i.e. kernel-based hardware-assisted virtualization, to establish a base line. Second, QEMU only uses software virtualization with single-threaded TCG, i.e. the QEMU tiny code generator that translates instructions of the guest architecture to instructions of the host architecture, which is a prerequisite for CoRReCt. The third scenario is CoRReCt, TCG run with record-replay and accessed through a comparator system. tables 4.2 and 4.3 show the results of an I/O-heavy

KVM	508 MB/s	±	1.20 %
TCG	155 MB/s	±	2.48 %
CoRReCt	23.7 MB/s	±	30.10 %

Table 4.2.: Average writing speed across three runs FIO with CoRReCt and compared to pure TCG and KVM as baseline

KVM	6.77 FPS	±	0.41 %
TCG	0.21 FPS	±	0.45 %
CoRReCt	0.20 FPS	±	0.24 %

Table 4.3.: Average achieved frames per second (higher numbers are better) across three runs for encoding video into h264 with ffmpeg. Then Benchmark is run with CoRReCt and compared to pure TCG and KVM as baseline

benchmark[7] using the Flexible IO Tester resp. the results of a CPU-heavy benchmark[8] that uses FFmpeg. The exact commands and dependencies used to execute these benchmarks in the VM can be found in the linked files.

In both cases, there is a significant performance decrease when disabling KVM, which is expected due to the overhead of TCG. Compared to TCG execution, using CoRReCt has a significant impact on I/O. On an abstract level, this stems from the fact that I/O timing involves a lot of non-deterministic behavior that has to be recorded and replayed. Optimizing parts of QEMU to handle this more efficiently could narrow this performance gap. Computation itself is only very slightly impacted by CoRReCt in comparison to TCG execution as it is mostly deterministic and the main overhead stems from the effort of re-playing events. To capture this accurately, the benchmark was extended to wait for a synchronization point after the computation finished, ensuring that the CoRReCt system is completely done processing this benchmark. The conclusion drawn from these results is that the main cost associated with using the CoRReCt system stems from running QEMU in TCG mode, especially for computation-heavy workloads. For I/O-bound tasks, CoRReCt incurs overhead due to synchronization. Reduced performance compared to hardware virtualization is an expected outcome, but necessary if different host architectures are to be used.

4.5. Conclusion

We presented an approach to improve the security and privacy of software executions on untrusted systems. To this end, we used virtual machines to execute the software, where non-deterministic events are recorded on one and replayed on another VM. Cryptographically-secure randomness is generated through a coin-toss, replacing randomness-harvesting methods that rely on non-determinism and may negatively impact security. Inputs and outputs are handled by a trusted interface and only allowed if they are the same by both VMs.

We provided a formal model for our approach, together with a proof of security in this model. Assuming a trusted input-output interface, our approach leads to an execution with strong security guarantees:

- If at most one system is corrupted, the correct inputs are used and either a correct result or no result at all is returned.
- If at most one system is corrupted and timing side-channels are not considered, then the additional leakage introduced by our approach is bounded by $\log_2(n)$ bits, where n is the

[7] https://openbenchmarking.org/test/pts/fio (Random Read/Write, Sync, No Direct, 8MB Block Size)
[8] https://openbenchmarking.org/test/pts/ffmpeg (h264, Live)

number of messages sent. If the recording system is honest and timing side-channels are considered, the same bound can be established.

Given that the VMs can be run on different host platforms, we believe that this assumption is very plausible. Special care was taken to model an adversary's influence on an otherwise honest execution through its choice of non-determinism.

We discussed several practically relevant applications of our approach and presented our implementation based on QEMU Linux VMs, demonstrating the practicality of CoRReCt.

5. Conclusion

5.1. Balance of Trust

The application examples have shown, that trust in a computer system is a resource that can be spent for performance benefits. From a cloud computing environment, where the required trust in the operator is the highest, over bare metal systems and CPU-vendor implemented enclaves used under varying trust assumptions to the most conservative trust assmumptions presented for CoRReCt (chapter 4) we see that less trust requires more security measures which have impact on performance. The challenge to strike a balance between trust and security measures is ultimately dependent on the specific application which brings in the concrete trust assumptions, that the operator is willing to make, and the concrete impact that an attack on the application might have.

5.2. Security Engineering

The two main focus areas when building and securing complex applications today are: vulnerabilities or design flaws in the application itself, and supply-chain attacks via an applications dependencies. Securing an application against supply chain attacks is primarily concerned with open source dependencies, in constrast to commercial dependencies. Due to the openness of the development model of open source dependencies, and often their huge quantity, attacks are more likely here, however they can also be detected more easily. For commercial dependencies there is usually not much more than complete trust into the supplier. This is true for software dependencies, but even more so for infrastructure providers.

Securing an application against a network attacker is conceptually straight forward, by using established building blocks, but increasing resilience against supply chain attacks, not just against a denial of service threat, but for general security threats, is more challenging. Increasing this resilience requires architectural decisions which might have some cost on other quality apsects of the application like performance, features and development effort.

5.3. Future Work

Tools for engineering secure applications often focus on a specific problem, like dependencies with known vulnerabilities, typical patterns that suggest vulnerabilities or tools that scan dynamically for typical misconfigurations. As all these tools individually have their shortcomings, security engineering often comes down to best-practices, which steer towards a layered approach. Individual security measures are accepted to not be perfect, and applications are engineered in a way to minimze impact if a measure fails. This leads to a system where the individual benefit of security measures is sometimes hard to estimate. We envision for security engineering, as it

continues to grow in importance, to gain a more solid theoretical foundation. This could lead to a dicipline that allows to evaluate the security of complex applications, and to evaluate the impact of security mesaures, or architecture decisions, so the secuity goals of the application can be ensured. Such a work is dependent on thoroughly understood building blocks and their interactions. For a more direct applicability of this research, more complex applications should be researched, considering a security model that is closer to the nuances of the environment that the application is running in.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org. 2015. url: https://www.tensorflow.org/.

[2] Mark Abspoel, Daniel Escudero, and Nikolaj Volgushev. "Secure training of decision trees with continuous attributes". In: Proceedings on Privacy Enhancing Technologies 2021.1 (Jan. 2021), pp. 167–187. doi: 10.2478/popets-2021-0010.

[3] Dirk Achenbach, Jörn Müller-Quade, and Jochen Rill. "Universally composable firewall architectures using trusted hardware". In: Cryptography and Information Security in the Balkans: First International Conference, BalkanCryptSec 2014, Istanbul, Turkey, October 16-17, 2014, Revised Selected Papers 1. Springer. 2015, pp. 57–74.

[4] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX". In: ISOC Network and Distributed System Security Symposium – NDSS 2018. San Diego, CA, USA: The Internet Society, Feb. 2018. doi: 10.14722/ndss.2018.23284.

[5] Adi Akavia, Max Leibovich, Yehezkel S. Resheff, Roey Ron, Moni Shahar, and Margarita Vald. "Privacy-Preserving Decision Trees Training and Prediction". In: ACM Trans. Priv. Secur. 25.3 (2022), 24:1–24:30. doi: 10.1145/3517197. url: https://doi.org/10.1145/3517197.

[6] Apple. Secure Enclave. last accessed on 31.08.2022. 2022. url: https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web.

[7] Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. "(If) Size Matters: Size-Hiding Private Set Intersection". In: PKC 2011: 14th International Conference on Theory and Practice of Public Key Cryptography. Ed. by Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi. Vol. 6571. Lecture Notes in Computer Science. Taormina, Italy: Springer Berlin Heidelberg, Germany, Mar. 2011, pp. 156–173. doi: 10.1007/978-3-642-19379-8_10.

[8] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. "Universal Composition with Global Subroutines: Capturing Global Setup Within Plain UC". In: TCC 2020: 18th Theory of Cryptography Conference, Part III. Ed. by Rafael Pass and Krzysztof Pietrzak. Vol. 12552. Lecture Notes in Computer Science. Durham, NC, USA: Springer, Cham, Switzerland, Nov. 2020, pp. 1–30. doi: 10.1007/978-3-030-64381-2_1.

[9] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. "Secure Multiparty Computation from SGX". In: FC 2017: 21st International Conference on Financial Cryptography and Data Security. Ed. by Aggelos Kiayias. Vol. 10322. Lecture Notes in Computer Science. Sliema, Malta: Springer, Cham, Switzerland, Apr. 2017, pp. 477–497. doi: 10.1007/978-3-319-70972-7_27.

[11] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. "A Concrete Security Treatment of Symmetric Encryption". In: 38th Annual Symposium on Foundations of Computer Science. Miami Beach, Florida: IEEE Computer Society Press, Oct. 1997, pp. 394–403. doi: 10.1109/SFCS.1997.646128.

[12] Candice Bentéjac, Anna Csörg, and Gonzalo Martnez-Muñoz. "A comparative analysis of gradient boosting algorithms". In: Artificial Intelligence Review 54 (2021), pp. 1937–1967.

[13] Robin Berger, Felix Dörre, and Alexander Koch. "Two-Party Decision Tree Training from Updatable Order-Revealing Encryption". In: International Conference on Applied Cryptography and Network Security. Springer. 2024, pp. 288–317. doi: 10.1007/978-3-031-54770-6_12.

[14] Pramod Bhatotia, Markulf Kohlweiss, Lorenzo Martinico, and Yiannis Tselekounis. "Steel: Composable Hardware-Based Stateful and Randomised Functional Encryption". In: PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II. Ed. by Juan Garay. Vol. 12711. Lecture Notes in Computer Science. Virtual Event: Springer, Cham, Switzerland, May 2021, pp. 709–736. doi: 10.1007/978-3-030-75248-4_25.

[15] Manuel Blum. "Coin flipping by telephone a protocol for solving impossible problems". In: ACM SIGACT News 15.1 (1983), pp. 23–27.

[16] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. "Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation". In: Advances in Cryptology – EUROCRYPT 2015, Part II. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. Lecture Notes in Computer Science. Sofia, Bulgaria: Springer Berlin Heidelberg, Germany, Apr. 2015, pp. 563–594. doi: 10.1007/978-3-662-46803-6_19.

[17] Dan Boneh and Victor Shoup. "A graduate course in applied cryptography". In: Draft 0.6 (2026).

[18] Thomas C Bressoud and Fred B Schneider. "Hypervisor-based fault tolerance". In: ACM Transactions on Computer Systems (TOCS) 14.1 (1996), pp. 80–107.

[19] Brandon Broadnax, Alexander Koch, Jeremias Mechler, Tobias Müller, Jörn Müller-Quade, and Matthias Nagel. "Fortified Multi-Party Computation: Taking Advantage of Simple Secure Hardware Modules". In: Proceedings on Privacy Enhancing Technologies 2021.4 (Oct. 2021), pp. 312–338. doi: 10.2478/popets-2021-0072.

[20] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. "API design for machine learning software: experiences from the scikit-learn project". In: ECML PKDD Workshop: Languages for Data Mining and Machine Learning. 2013, pp. 108–122.

[21] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. "The Wonderful World of Global Random Oracles". In: Advances in Cryptology – EUROCRYPT 2018, Part I. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10820. Lecture Notes in Computer Science. Tel Aviv, Israel: Springer, Cham, Switzerland, Apr. 2018, pp. 280–312. doi: 10.1007/978-3-319-78381-9_11.

[22] Ran Canetti. "Universally Composable Security". In: J. ACM 67.5 (2020), 28:1–28:94. doi: 10.1145/3402457. url: https://doi.org/10.1145/3402457.

[23] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: 42nd Annual Symposium on Foundations of Computer Science. Las Vegas, NV, USA: IEEE Computer Society Press, Oct. 2001, pp. 136–145. doi: 10.1109/SFCS.2001.959888.

[24] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. "Universally Composable Security with Global Setup". In: TCC 2007: 4th Theory of Cryptography Conference. Ed. by Salil P. Vadhan. Vol. 4392. Lecture Notes in Computer Science. Amsterdam, The Netherlands: Springer Berlin Heidelberg, Germany, Feb. 2007, pp. 61–85. doi: 10.1007/978-3-540-70936-7_4.

[25] Ran Canetti and Marc Fischlin. "Universally Composable Commitments". In: Advances in Cryptology – CRYPTO 2001. Ed. by Joe Kilian. Vol. 2139. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2001, pp. 19–40. doi: 10.1007/3-540-44647-8_2.

[26] Ran Canetti, Oded Goldreich, and Shai Halevi. "On the Random-Oracle Methodology as Applied to Length-Restricted Signature Schemes". In: TCC 2004: 1st Theory of Cryptography Conference. Ed. by Moni Naor. Vol. 2951. Lecture Notes in Computer Science. Cambridge, MA, USA: Springer Berlin Heidelberg, Germany, Feb. 2004, pp. 40–57. doi: 10.1007/978-3-540-24638-1_3.

[27] Ran Canetti, Oded Goldreich, and Shai Halevi. "The Random Oracle Methodology, Revisited (Preliminary Version)". In: 30th Annual ACM Symposium on Theory of Computing. Dallas, TX, USA: ACM Press, May 1998, pp. 209–218. doi: 10.1145/276698.276741.

[28] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. "Practical UC security with a Global Random Oracle". In: ACM CCS 2014: 21st Conference on Computer and Communications Security. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. Scottsdale, AZ, USA: ACM Press, Nov. 2014, pp. 597–608. doi: 10.1145/2660267.2660374.

[29] Ran Canetti and Hugo Krawczyk. "Universally Composable Notions of Key Exchange and Secure Channels". In: Advances in Cryptology – EUROCRYPT 2002. Ed. by Lars R. Knudsen. Vol. 2332. Lecture Notes in Computer Science. Amsterdam, The Netherlands: Springer Berlin Heidelberg, Germany, Apr. 2002, pp. 337–351. doi: 10.1007/3-540-46035-7_22.

[30] J.Lawrence Carter and Mark N. Wegman. "Universal classes of hash functions". In: Journal of Computer and System Sciences 18.2 (1979), pp. 143–154. issn: 0022-0000. doi: https://doi.org/10.1016/0022-0000(79)90044-8. url: https://www.sciencedirect.com/science/article/pii/0022000079900448.

[31] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999. Ed. by Margo I. Seltzer and Paul J. Leach. USENIX Association, 1999, pp. 173–186. url: https://dl.acm.org/citation.cfm?id=296824.

[32] David Challener, Kent Yoder, Ryan Catherman, David Safford, and Leendert Van Doorn. A practical guide to trusted computing. Pearson Education, 2007.

[33] Kamalika Chaudhuri and Claire Monteleoni. "Privacy-preserving logistic regression". In: Advances in neural information processing systems 21 (2008).

[34] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system". In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 2016, pp. 785–794.

[35] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. "Practical Order-Revealing Encryption with Limited Leakage". In: Fast Software Encryption – FSE 2016. Ed. by Thomas Peyrin. Vol. 9783. Lecture Notes in Computer Science. Bochum, Germany: Springer Berlin Heidelberg, Germany, Mar. 2016, pp. 474–493. doi: 10.1007/978-3-662-52993-5_24.

[36] Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V. L. Pereira. "SortingHat: Efficient Private Decision Tree Evaluation via Homomorphic Encryption and Transciphering". In: ACM CCS 2022: 29th Conference on Computer and Communications Security. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 563–577. doi: 10.1145/3548606.3560702.

[37] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. 2016. url: https://eprint.iacr.org/2016/086.

[38] Victor Costan and Srinivas Devadas. "Intel SGX explained". In: Cryptology ePrint Archive (2016).

[39] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography". In: IEEE Transactions on Information Theory 22.6 (1976), pp. 644–654. doi: 10.1109/TIT.1976.1055638.

[41] Felix Dörre, Marco Liebel, Jeremias Mechler, and Jörn Müller-Quade. "CoRReCt: Compute, Record, Replay, Compare to Secure Computations on Untrusted Systems". 2025.

[42] Felix Dörre, Jeremias Mechler, and Jörn Müller-Quade. "Practically Efficient Private Set Intersection from Trusted Hardware with Side-Channels". In: Advances in Cryptology – ASIACRYPT 2023. Ed. by Jian Guo and Ron Steinfeld. Singapore: Springer Nature Singapore, 2023, pp. 268–301. isbn: 978-981-99-8730-6. doi: 10.1007/978-981-99-8730-6_9.

[43] Wenliang Du and Zhijun Zhan. "Building decision tree classifier on private data". In: (2002).

[44] Thai Duong, Duong Hieu Phan, and Ni Trieu. "Catalic: Delegated PSI Cardinality with Applications to Contact Tracing". In: Advances in Cryptology – ASIACRYPT 2020, Part III. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12493. Lecture Notes in Computer Science. Daejeon, South Korea: Springer, Cham, Switzerland, Dec. 2020, pp. 870–899. doi: 10.1007/978-3-030-64840-4_29.

[45] F. Betül Durak, Thomas M. DuBuisson, and David Cash. "What Else is Revealed by Order-Revealing Encryption?" In: ACM CCS 2016: 23rd Conference on Computer and Communications Security. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. Vienna, Austria: ACM Press, Oct. 2016, pp. 1155–1166. doi: 10.1145/2976749.2978379.

[46] Jordan Frery, Andrei Stoian, Roman Bredehoft, Luis Montero, Celia Kherfallah, Benoît Chevallier-Mames, and Arthur Meyre. Privacy-Preserving Tree-Based Inference with Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2023/258. 2023. url: https://eprint.iacr.org/2023/258.

[47] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. "Oblivious Key-Value Stores and Amplification for Private Set Intersection". In: Advances in Cryptology – CRYPTO 2021, Part II. Ed. by Tal Malkin and Chris Peikert. Vol. 12826. Lecture Notes in Computer Science. Virtual Event: Springer, Cham, Switzerland, Aug. 2021, pp. 395–425. doi: 10.1007/978-3-030-84245-1_14.

[48] Scott Garriss, Michael Kaminsky, Michael J. Freedman, Brad Karp, David Mazières, and Haifeng Yu. "RE: Reliable Email". In: 3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings. Ed. by Larry L. Peterson and Timothy Roscoe. USENIX, 2006. url: http://www.usenix.org/events/nsdi06/tech/garriss.html.

[49] Oded Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs". In: 19th Annual ACM Symposium on Theory of Computing. Ed. by Alfred Aho. New York City, NY, USA: ACM Press, May 1987, pp. 182–194. doi: 10.1145/28395.28416.

[50] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA. Ed. by Alfred V. Aho. 1987, pp. 218–229. doi: 10.1145/28395.28420. url: https://doi.org/10.1145/28395.28420.

[51] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. "Leakage-Abuse Attacks against Order-Revealing Encryption". In: 2017 IEEE Symposium on Security and Privacy (SP). 2017, pp. 655–672. doi: 10.1109/SP.2017.44.

[52] Shay Gueron. Intel advanced encryption standard (AES) new instructions set. 2010.

[53] Koki Hamada, Dai Ikarashi, Ryo Kikuchi, and Koji Chida. "Efficient decision tree training with new data structure for secure multi-party computation". In: Proceedings on Privacy Enhancing Technologies 2023.1 (Jan. 2023), pp. 343–364. doi: 10.56553/popets-2023-0021.

[54] Danny Harnik, Joe Kilian, Moni Naor, Omer Reingold, and Alon Rosen. "On robust combiners for oblivious transfer and other primitives". In: Advances in Cryptology–EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005. Proceedings 24. Springer. 2005, pp. 96–113.

[55] Carmit Hazay and Yehuda Lindell. "Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries". In: TCC 2008: 5th Theory of Cryptography Conference. Ed. by Ran Canetti. Vol. 4948. Lecture Notes in Computer Science. San Francisco, CA, USA: Springer Berlin Heidelberg, Germany, Mar. 2008, pp. 155–175. doi: 10.1007/978-3-540-78524-8_10.

[56] Amir Herzberg. "Folklore, practice and theory of robust combiners". In: Journal of Computer Security 17.2 (2009), pp. 159–189.

[57] Sebastiaan de Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker. "Practical Secure Decision Tree Learning in a Teletreatment Application". In: FC 2014: 18th International Conference on Financial Cryptography and Data Security. Ed. by Nicolas Christin and Reihaneh Safavi-Naini. Vol. 8437. Lecture Notes in Computer Science. Christ Church, Barbados: Springer Berlin Heidelberg, Germany, Mar. 2014, pp. 179–194. doi: 10.1007/978-3-662-45472-5_12.

[58] IBM. IBM Secure Execution for Linux. last accessed on 31.08.2022. 2022. url: https://www.ibm.com/downloads/cas/O158MBWG.

[59] Intel. Intel Software Guard Extensions (Intel SGX). 2023. url: https://download.01.org/intel-sgx/sgx-linux/2.9.1/docs/Intel_SGX_Developer_Guide.pdf.

[60] Xabier Iturbe, Balaji Venu, Emre Ozer, and Shidhartha Das. "A triple core lock-step (TCLS) ARMő Cortexő-R5 processor for safety-critical and ultra-reliable applications". In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W). IEEE. 2016, pp. 246–249.

[61] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. "Intel software guard extensions: EPID provisioning and attestation services". In: White Paper 1.1-10 (2016), p. 119.

[62] Mireya Jurado, Catuscia Palamidessi, and Geoffrey Smith. "A Formal Information-Theoretic Leakage Analysis of Order-Revealing Encryption". In: CSF 2021: IEEE 34th Computer Security Foundations Symposium. Ed. by Ralf Küsters and Dave Naumann. Virtual Conference: IEEE Computer Society Press, June 2021, pp. 1–16. doi: 10.1109/CSF51468.2021.00046.

[63] Marcel Keller. "MP-SPDZ: A Versatile Framework for Multi-Party Computation". In: ACM CCS 2020: 27th Conference on Computer and Communications Security. 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.

[64] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. "Crypten: Secure multi-party computation meets machine learning". In: Advances in Neural Information Processing Systems 34 (2021), pp. 4961–4973.

[65] Miroslav Kubat and JA Kubat. An introduction to machine learning. Vol. 2. Springer, 2017.

[66] Anunay Kulshrestha and Jonathan R. Mayer. "Identifying Harmful Media in End-to-End Encrypted Communication: Efficient Private Membership Computation". In: 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 893–910. url: https://www.usenix.org/conference/usenixsecurity21/presentation/kulshrestha.

[67] Arvind Kumar. Active platform management demystified: unleashing the power of intel VPro (TM) technology. Intel Press, 2009.

[68] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. "The Byzantine Generals Problem". In: ACM Trans. Program. Lang. Syst. 4.3 (1982), pp. 382–401. doi: 10.1145/357172.357176. url: https://doi.org/10.1145/357172.357176.

[69] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network". In: IEEE Access 10 (2022), pp. 30039–30054.

[70] Kevin Lewi and David J. Wu. "Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds". In: ACM CCS 2016: 23rd Conference on Computer and Communications Security. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. Vienna, Austria: ACM Press, Oct. 2016, pp. 1167–1178. doi: 10.1145/2976749.2978376.

[71] Yuan Li, Hongbing Wang, and Yunlei Zhao. "Delegatable Order-Revealing Encryption". In: ASIACCS 19: 14th ACM Symposium on Information, Computer and Communications Security. Ed. by Steven D. Galbraith, Giovanni Russello, Willy Susilo, Dieter Gollmann, Engin Kirda, and Zhenkai Liang. Auckland, New Zealand: ACM Press, July 2019, pp. 134–147. doi: 10.1145/3321705.3329829.

[72] Marco Liebel. "Überprüfung von Schutzzielen der Informationssicherheit durch Aufzeichnung und Wiedergabe virtueller Maschinen". 2022. url: https://dx.doi.org/10.5445/IR/1000149965.

[73] Yehuda Lindell and Benny Pinkas. "Privacy Preserving Data Mining". In: Advances in Cryptology – CRYPTO 2000. Ed. by Mihir Bellare. Vol. 1880. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2000, pp. 36–54. doi: 10.1007/3-540-44598-6_3.

[74] Haikun Liu, Hai Jin, Xiaofei Liao, and Zhengqiu Pan. "XenLR: Xen-based Logging for Deterministic Replay". In: 2008 Japan-China Joint Workshop on Frontier of Computer Science and Technology. 2008, pp. 149–154. doi: 10.1109/FCST.2008.31.

[75] Ximeng Liu, Robert H. Deng, Kim-Kwang Raymond Choo, and Jian Weng. "An Efficient Privacy-Preserving Outsourced Calculation Toolkit With Multiple Keys". In: IEEE Transactions on Information Forensics and Security 11.11 (2016), pp. 2401–2414. doi: 10.1109/TIFS.2016.2573770.

[76] Yibiao Lu, Bingsheng Zhang, Hong-Sheng Zhou, Weiran Liu, Lei Zhang, and Kui Ren. "Correlated Randomness Teleportation via Semi-trusted Hardware - Enabling Silent Multi-party Computation". In: ESORICS 2021: 26th European Symposium on Research in Computer Security, Part II. Ed. by Elisa Bertino, Haya Shulman, and Michael Waidner. Vol. 12973. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Cham, Switzerland, Oct. 2021, pp. 699–720. doi: 10.1007/978-3-030-88428-4_34.

[77] Chunyang Lv, Jianfeng Wang, Shi-Feng Sun, Yunling Wang, Saiyu Qi, and Xiaofeng Chen. "Towards Practical Multi-Client Order-Revealing Encryption: Improvement and Application". In: IEEE Transactions on Dependable and Secure Computing (2023).

[78] Moxie Marlinspike. Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/. 2017.

[79] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. "Flicker: an execution infrastructure for tcb minimization". In: Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008. Ed. by Joseph S. Sventek and Steven Hand. 2008, pp. 315–328. doi: 10.1145/1352592.1352625. url: https://doi.org/10.1145/1352592.1352625.

[80] Damiano Melotti, Maxime Rossi-Bellom, and Andrea Continella. “Reversing and Fuzzing the Google Titan M Chip”. In: Reversing and Offensive-oriented Trends Symposium. 2021, pp. 1–10.

[81] Jörn Müller-Quade and Dominique Unruh. “Long-Term Security and Universal Composability”. In: Journal of Cryptology 23.4 (Oct. 2010), pp. 594–671. doi: 10.1007/s00145-010-9068-8.

[82] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX”. In: 2020 IEEE Symposium on Security and Privacy. San Francisco, CA, USA: IEEE Computer Society Press, May 2020, pp. 1466–1482. doi: 10.1109/SP40000.2020.00057.

[83] Moni Naor, Benny Pinkas, and Omer Reingold. “Distributed Pseudo-random Functions and KDCs”. In: Advances in Cryptology – EUROCRYPT’99. Ed. by Jacques Stern. Vol. 1592. Lecture Notes in Computer Science. Prague, Czech Republic: Springer Berlin Heidelberg, Germany, May 1999, pp. 327–346. doi: 10.1007/3-540-48910-X_23.

[84] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. “Reproducible Containers”. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 167–182. isbn: 9781450371025. doi: 10.1145/3373376.3378519. url: https://doi.org/10.1145/3373376.3378519.

[85] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. “A Survey of Published Attacks on Intel SGX”. In: CoRR abs/2006.13598 (2020). arXiv: 2006.13598. url: https://arxiv.org/abs/2006.13598.

[86] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. “Oblivious Multi-Party Machine Learning on Trusted Processors”. In: USENIX Security 2016: 25th USENIX Security Symposium. Ed. by Thorsten Holz and Stefan Savage. Austin, TX, USA: USENIX Association, Aug. 2016, pp. 619–636.

[87] Markus Ottela. Tinfoil Chat-Instant messaging with endpoint security. 2017. url: https://github.com/maqp/tfc/.

[88] Rafael Pass, Elaine Shi, and Florian Tramèr. “Formal Abstractions for Attested Execution Secure Processors”. In: Advances in Cryptology – EUROCRYPT 2017, Part I. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. Paris, France: Springer, Cham, Switzerland, Apr. 2017, pp. 260–289. doi: 10.1007/978-3-319-56620-7_10.

[89] Krzysztof Pietrzak. “Non-trivial black-box combiners for collision-resistant hash-functions dont exist”. In: Advances in Cryptology-EUROCRYPT 2007: 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007. Proceedings 26. Springer. 2007, pp. 23–33.

[90] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. “PSI from PaXoS: Fast, Malicious Private Set Intersection”. In: Advances in Cryptology – EUROCRYPT 2020, Part II. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. Lecture Notes in Computer Science. Zagreb, Croatia: Springer, Cham, Switzerland, May 2020, pp. 739–767. doi: 10.1007/978-3-030-45724-2_25.

[91] J Ross Quinlan. C4. 5: programs for machine learning. Elsevier, 2014.

[92] Peter Rindal and Srinivasan Raghuraman. “Blazing Fast PSI from Improved OKVS and Subfield VOLE”. In: IACR Cryptol. ePrint Arch. (2022), p. 320. url: https://eprint.iacr.org/2022/320.

[93] Peter Rindal and Mike Rosulek. “Improved Private Set Intersection Against Malicious Adversaries”. In: Advances in Cryptology – EUROCRYPT 2017, Part I. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. Paris, France: Springer, Cham, Switzerland, Apr. 2017, pp. 235–259. doi: 10.1007/978-3-319-56620-7_9.

[94] Michiel Ronsse and Koen De Bosschere. “RecPlay: A fully integrated practical record/replay system”. In: ACM Transactions on Computer Systems (TOCS) 17.2 (1999), pp. 133–152.

[95] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. “The design of a practical system for fault-tolerant virtual machines”. In: ACM SIGOPS Operating Systems Review 44.4 (2010), pp. 30–39.

[96] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: ACM CCS 2019: 26th Conference on Computer and Communications Security. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. London, UK: ACM Press, Nov. 2019, pp. 753–768. doi: 10.1145/3319535.3354252.

[97] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. “Jalangi: A selective record-replay and dynamic analysis framework for JavaScript”. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. 2013, pp. 488–498.

[98] Emmanuel Stapf, Patrick Jauernig, Ferdinand Brasser, and Ahmad-Reza Sadeghi. “In Hardware We Trust? From TPM to Enclave Computing on RISC-V”. In: 29th IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2021, Singapore, Singapore, October 4-7, 2021. IEEE, 2021, pp. 1–6. doi: 10.1109/VLSI-SoC53125.2021.9606968. url: https://doi.org/10.1109/VLSI-SoC53125.2021.9606968.

[99] Hao Sun, Jinshu Su, Xiaofeng Wang, Rongmao Chen, Yujing Liu, and Qiaolin Hu. “PriMal: Cloud-Based Privacy-Preserving Malware Detection”. In: ACISP 17: 22nd Australasian Conference on Information Security and Privacy, Part II. Ed. by Josef Pieprzyk and Suriadi Suriadi. Vol. 10343. Lecture Notes in Computer Science. Auckland, New Zealand: Springer, Cham, Switzerland, July 2017, pp. 153–172. doi: 10.1007/978-3-319-59870-3_9.

[100] Sandeep Tamrakar, Jian Liu, Andrew Paverd, Jan-Erik Ekberg, Benny Pinkas, and N. Asokan. “The Circle Game: Scalable Private Membership Test Using Trusted Hardware”. In: ASIACCS 17: 12th ACM Symposium on Information, Computer and Communications Security. Ed. by Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi. Abu Dhabi, United Arab Emirates: ACM Press, Apr. 2017, pp. 31–44. doi: 10.1145/3052973.3053006.

[101] Suryakanthi Tangirala. “Evaluating the impact of GINI index and information gain on classification using decision tree classifier algorithm”. In: International Journal of Advanced Computer Science and Applications 11.2 (2020), pp. 612–619.

[102] Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. “Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge”. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017. IEEE, 2017, pp. 19–34. doi: 10.1109/EuroSP.2017.28. url: https://doi.org/10.1109/EuroSP.2017.28.

[103] Jaideep Vaidya, Chris Clifton, Murat Kantarcioglu, and A. Scott Patterson. “Privacy-preserving decision trees over vertically partitioned data”. In: ACM Trans. Knowl. Discov. Data 2.3 (2008), 14:1–14:27. doi: 10.1145/1409620.1409624. url: https://doi.org/10.1145/1409620.1409624.

[104] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: USENIX Security 2018: 27th USENIX Security Symposium. Ed. by William Enck and Adrienne Porter Felt. Baltimore, MD, USA: USENIX Association, Aug. 2018, pp. 991–1008.

[105] Andrew Chi-Chih Yao. “How to Generate and Exchange Secrets (Extended Abstract)”. In: 27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986. IEEE Computer Society, 1986, pp. 162–167. doi: 10.1109/SFCS.1986.25. url: https://doi.org/10.1109/SFCS.1986.25.

[106] Jun Zhu, Wei Dong, Zhefu Jiang, Xiaogang Shi, Zhen Xiao, and Xiaoming Li. “Improving the performance of hypervisor-based fault tolerance”. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE. 2010, pp. 1–10.