Decentralizing Software Identity Management

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

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

genehmigte

Dissertation

von

Oliver Stengele

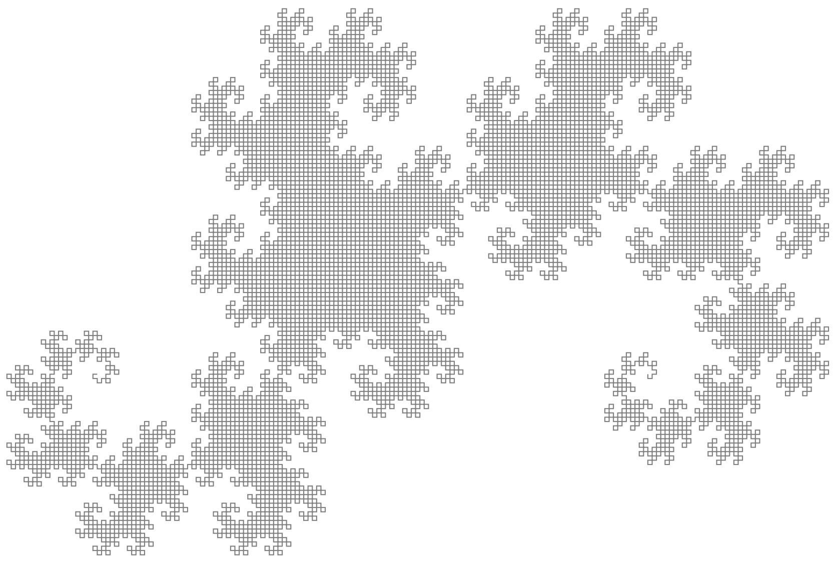
aus Heidelberg

Tag der mündlichen Prüfung: 22. Mai 2023
Erster Gutachter: Prof. Dr. rer. nat. Hannes Hartenstein
Zweiter Gutachter: Prof. Dr.-Ing. Stefan Tai

This work was supported in part by the German Federal Ministry of Education and Research within the project *KASTEL_ISE* in the Competence Center for Applied Security Technology (*KASTEL*), by the Helmholtz Association (HGF) via the topic *Engineering Secure Systems*, and by the German Federal Ministry for Economic Affairs and Climate Action within the research project Software-Defined Car (*SofDCar*).

The loss of illusions and the discovery of identity, though painful at first, can be ultimately exhilarating and strengthening.

Abraham Maslow



Abstract

Software holds a place of crucial importance in various contexts: business, commerce, industrial control systems, transportation, logistics, communication, and personal use to name a few. Consequently, it is vital to obtain software with integrity and an explicit endorsement from their authoritative source, like their developer or publisher. In this work, we endeavor to make the interaction between software creators and users more secure through the establishment and use of explicit identities for software. At the most basic level, a software identity establishes a unique and persistent reference point to which creators of software can attach and remove information about binaries of their software. The ability to remove previously released binaries from a software identity enables developers to react to security-critical bugs or compromise by clearly stating that these binaries should no longer be used. Users of a particular software can be informed about both revocations and new releases by monitoring the corresponding software identity and use it to verify the integrity and endorsement of newly obtained binaries.

Distributed ledger technologies like Ethereum, or Bitcoin before it, appear to be viable platforms to realize software identities without relying on a centrally trusted provider. An open peer-to-peer network establishes consensus on a tamper-proof state history, the eponymous ledger, and provides access to said history on demand. Ethereum is one of the earliest distributed ledgers to enable *smart contracts*, programs deployed to a distributed ledger that establish and manage a uniquely referenceable part of the overall ledger state. Crucially, the programming of smart contracts determines how, when, and by whom this substate can be manipulated.

The first research question of this dissertation aims to explore the viability of distributed ledger technologies to establish, manage, and use software identities. In particular, we investigate how useful properties for software identity management and use can be derived from the security properties provided by distributed ledgers in conjunction with further assumptions.

In addition to using software identities to augment and secure the distribution of software, we also explore their use as the basis for independent reviews of software releases. Executing such reviews on distributed ledgers presents a challenge regarding the disclosure of results. At the time of writing, no distributed ledger offers a corresponding functionality to both execute a procedure and document as a result that a set of statements were recorded independently. The second research question we pursue in this work therefore concerns the realization of such a disclosure mechanism on distributed ledgers based on existing cryptographic primitives.

We approach both research questions by designing, implementing, and evaluating corresponding decentralized applications on Ethereum as the most prominent instance of smart-contract-capable distributed ledgers. More specifically, we measure deployment and execution costs for smart contracts that comprise our decentralized applications to determine their practical viability. In two instances, we also evaluate computational effort that arises outside the ledger. We also semi-formally describe how the security properties of our proof of concept implementations can be derived from the underlying ledger and further assumptions.

We find Ethereum as an instance of smart-contract-capable distributed ledgers to be a viable platform for establishing and using software identities, including the aforementioned independent reviews. As our software identity management concept relies on rather fundamental properties of distributed ledgers, it should generalize well to systems other than Ethereum. By contrast, our concept for coordinated information disclosure relies on support for cryptographic operations on the ledger itself, which limits its generalizability. The costs to deploy the required smart contracts are significantly higher than the execution costs in typical use, which informs our recommendation for future work to improve the reusability of deployed contract instances. Regarding the coordinated disclosure of independently generated statements on a distributed ledger, we achieve overall cost reductions of 20–40 % compared to related work by exploiting differing cryptographic requirements. Our approach to realizing coordinated information disclosure on Ethereum relies on elliptic curve operations which, while sufficient, are rather limited at the time of writing. As such, our work adds to the rationale for expanding the set of supported elliptic curve in the future development of Ethereum.

Zusammenfassung

Software ist in unterschiedlichsten Bereichen von größter Wichtigkeit: Wirtschaft, Handel, Industrielle Steueranlagen, Transport, Logistik, Kommunikation, sowie im privaten Gebrauch um nur einige Beispiele zu nennen. Es ist entsprechend unverzichtbar, Software mit Integrität und einer expliziten Befürwortung durch den jeweiligen Entwickler oder Herausgeber zu beziehen. In dieser Arbeit verfolgen wir das Ziel, die Interaktion zwischen Erstellern und Nutzern von Software durch die Etablierung und Nutzung von expliziten Identitäten für Software weiter abzusichern. Eine Softwareidentität etabliert in erster Linie einen eindeutigen und persistenten Bezugspunkt an den Softwareersteller Informationen zu Binärdateien ihrer Software anhängen und entfernen können. Die Möglichkeit zuvor veröffentlichte Binärdateien aus einer Softwareidentität zu entfernen erlaubt Entwicklern auf sicherheitskritische Fehler oder Kompromittierungen zu reagieren, indem sie klar kommunizieren, dass bestimmte Binärdateien nicht länger verwendet werden sollten. Nutzer einer Software können über solche Widerrufe oder neue Versionen informiert werden, indem sie die entsprechende Softwareidentität beobachten über die sie dann auch die Integrität und Befürwortung von heruntergeladenen Binärdateien überprüfen können.

Distributed Ledger Technologien wie Ethereum oder zuvor Bitcoin scheinen taugliche Plattformen für die Umsetzung von Softwareidentitäten zu sein, ohne dabei auf zentrale Anbieter vertrauen zu müssen. Ein offenes Peer-to-Peer Netzwerk etabliert einen Konsens über einen manipulationsgeschützten Zustandsverlauf, der namensgebende Ledger, und ermöglicht Zugriff auf selbigen. Ethereum ist einer der ersten Distributed Ledger, der sogenannte *Smart Contracts* ermöglicht. Dabei handelt es sich um Programme, die auf einem Distributed Ledger installiert und ausgeführt werden und damit einen eindeutig referenzierbaren Teil des Ledgerzustandes etablieren und verwalten. Einzig und allein die Programmierung eines Smart Contracts bestimmt darüber, wer den Teilzustand wann und wie verändern kann.

Die erste Forschungsfrage dieser Dissertation zielt auf die Tauglichkeit von Distributed Ledger Technologien hinsichtlich der Etablierung, Verwaltung, und Nutzung von Softwareidentitäten ab. Insbesondere untersuchen wir, wie nützliche Eigenschaften für Softwareidentitätsmanagement und -nutzung von den Sicherheitseigenschaften des zugrundeliegenden Distributed Ledgers und weiteren Annahmen abgeleitet werden können.

Neben der Verwendung von Softwareidentitäten zur weiteren Absicherung der Softwaredistribution untersuchen wir außerdem ihre Nutzbarkeit als Grundlage für unabhängige Begutachtungen von Softwareversionen. Die Durchführung solcher unabhängigen Begutachtungen mittels Distributed Ledgern führt unweigerlich zu einer Herausforderung hinsichtlich der koordinierten Offenlegung der Ergebnisse. Zum Zeitpunkt der Abfassung dieser Arbeit bietet kein Distributed Ledger eine entsprechende Funktionalität, um die Erstellung einer Menge unabhängig erstellter Aussagen zu unterstützen oder zu dokumentieren. Die zweite Forschungsfrage dieser Arbeit befasst sich deshalb mit der Umsetzung eines Offenlegungsmechanismus für Distributed Ledger basierend auf bestehenden kryptografischen Primitiven.

Wir behandeln beide Forschungsfragen, indem wir entsprechende dezentrale Anwendungen konzipieren, implementieren, und evaluieren. Wir nutzen dabei Ethereum als prominentestes Exemplar eines Smart-Contract-fähigen Distributed Ledgers. Genauer gesagt messen wir die Installations- und Ausführungskosten jener Smart Contracts, die für unsere dezentralen Anwendungen nötig sind, um ihre praktische Tauglichkeit zu bestimmen. In zwei Fällen ermitteln wir außerdem den Rechenaufwand, der abseits des Ledgers anfällt. Wir zeigen zudem semi-formal, wie die Sicherheitseigenschaften unserer Proof of Concept Implementierung von dem zugrundeliegenden Distributed Ledger und weiteren Annahmen abgeleitet werden können.

Wir kommen zu dem Ergebnis, dass Ethereum stellvertretend für Smart-Contract-fähige Distributed Ledger eine taugliche Plattform für die Umsetzung von Softwareidentitäten ist, inklusive der zuvor angemerkten unabhängigen Begutachtungen. Da unser Konzept des Softwareidentitätsmanagements auf eher grundlegenden Eigenschaften von Distributed Ledgern fußt sollte es sich gut auf andere Systeme übertragen lassen. Im Gegensatz dazu erfordert unser Konzept für einen Offenlegungsmechanismus die Unterstützung von bestimmten kryptografischen Operationen auf dem verwendeten Ledger, was die Übertragbarkeit entsprechend einschränkt. Die Kosten für die Installation der nötigen Smart Contracts sind signifikant größer als die Ausführungskosten im typischen Gebrauch, weshalb wir für zukünftige Arbeit empfehlen, die Wiederverwendbarkeit von installierten Smart Contract Instanzen zu verbessern. Bei der koordinierten Offenlegung von unabhängig erstellten Aussagen auf einem Distributed Ledger erzielen wir eine Reduktion der Gesamtkosten von 20–40 % im Vergleich zu verwandter Arbeit, indem wir unterschiedliche kryptografische Anforderungen ausnutzen. Unser Ansatz um eine koordinierte Offenlegung auf Ethereum zu erzielen stützt sich auf Elliptische-Kurven-Operationen die, obwohl ausreichend, zum aktuellen Zeitpunkt sehr eingeschränkt sind. Entsprechend trägt unsere Arbeit einen weiteren Grund für die Erweiterung der unterstützten elliptischen Kurven im Zuge der Weiterentwicklung von Ethereum bei.

Contents

List of Figures

List of Tables

1. Introduction

Information technology had and continues to have a profound impact on human society at a global scale. In conjunction with the Internet, computers in the form of laptops, servers, smartphones, or tablets, and particularly the software that runs on them, have expanded the capabilities of humans to communicate, collaborate, and accomplish a wide variety of tasks. Consequently, software in general has attained critical importance for individuals as well as businesses and governments, as frequent cyberattacks[1] demonstrate. The relationship between creators and users of software, and specifically the task of ensuring that software in use is both genuine and secure, is becoming increasingly important.

For the purpose of this work, any particular software can be understood as a collection of digital objects, i.e. executable binaries, created by a developer for specific execution environments like operating systems and device types. Software in active development changes over time with the release of updates as features are added and mistakes are corrected, leading to new binaries being added to such a collection. For each software, a corresponding developer acts as an authoritative figure by expressing in some way which binaries belong to it. In this work, we introduce the notion of explicit *software identities* to clarify the affiliation between various binaries of a given software across both versions and target platforms. Essentially, a software identity establishes a unique reference point that remains constant even as the software evolves through continual development and new binaries of it are released.

As software is created by individuals or small groups of developers but can potentially be used by millions of users, a peculiar and very asymmetric trust relation can form. By using a software, users implicitly trust its creators to not have included malicious or destructive functionality, for example. Likewise, a responsible software developer should be interested in protecting users of their software from risks by informing them of security-critical updates, for example. In recent years, supply-chain attacks in particular have demonstrated the need for a reliable way to inform users of a given software version to perform updates or take other actions.[2] With commercial software, developers may be able to contact their customers directly in such cases. But particularly for freely available open-source software, direct contact between developers and users may not be possible or reliable, thus impeding emergency communications.

[1] For example, in May 2021, a major fuel pipeline in the U.S. had to be shut down due to a ransomware attack: `https://edition.cnn.com/2021/05/08/politics/colonial-pipeline-cybersecurity-attack/`

[2] In September 2017, a version of the popular Windows maintenance utility CCleaner was surreptitiously infected with a Trojan horse malware which spread to 2.27 million systems: `https://blog.avast.com/update-to-the-ccleaner-5.33.6162-security-incident`

Large IT companies like Microsoft, Apple, or Google employ infrastructure to ensure that users obtain software and updates with authenticity and integrity. In 2009, the decentralized cryptocurrency Bitcoin demonstrated how an open and dynamic group of mutually distrustful parties can indeed establish and maintain consensus over an append-only data structure later coined a *blockchain*. The following years saw the rise of numerous variations and iterations of the distributed ledger concept that Bitcoin pioneered, which are nowadays collectively referred to as *distributed ledger technologies* (DLT). One such iteration is *Ethereum* which was proposed in 2013 by Vitalik Buterin as a more general blockchain-based platform for decentralized applications and started operations in 2015. As public infrastructure that is not under the control of any single authority but available to everyone for the publication and retrieval of information, Ethereum presents an opportunity for developers to establish and manage identities of their software through stateful programs recorded on the Ethereum blockchain called *smart contracts*. Additionally, systems like Ethereum could also serve as a platform to conduct and document review processes for software releases in order to potentially discover bugs and mistakes or to attest verified attributes of the software before its wide-spread deployment. In this way, data related to these software identities can be stored in a highly available, uniquely referenceable, and tamper-resistant manner that does not rely on a centralized trusted third party.

Existing approaches to ensuring the authenticity, integrity, and creator endorsement of software binaries that rely on a public key infrastructure (PKI) exhibit shortcomings, particularly regarding the persistence, timeliness, and efficacy of revocations [48, 49]. Likewise, work on software certification [46, 71] focuses more on the certification process itself and less on how certification results can be attached to software or retreived afterwards. Denney [23] and Heck [41] note in their work the relevance of establishing such links. With software identity management, we aim to contribute a possible solution to both of the above issues by enabling precise revocations for binaries and establishing a persistent point of reference across binaries. By utilizing distributed ledgers as the basis for software identity management, we avoid depending on centralized providers or repository managers that have been the victims of attacks [67] and deceits[3].

As many other IT platforms, Ethereum has a symbiotic and bidirectional relationship with applications: Only through varied and useful applications does the platform thrive and grow, whereas without a platform, applications would not be possible. Just like the capabilities of the platform inform what applications can be built upon it, so too can the demands of applications inform the future development of the platform. In this dissertation, we aim to contribute to both halves of this feedback loop by exploring the current capabilities of Ethereum at the use case of identity management and subsequent review of software as depicted in Figure 1.1 while also deriving recommendations for the future development and operation of Ethereum from the gained insights. More specifically, our goal is to secure the distribution of software against malicious actors by ensuring the authenticity, integrity, and creator endorsement of binaries via a distributed ledger.

[3] In 2017, a fraudulent application disguised as an update to the widely-used messenger WhatsApp was published to Google's Play Store and downloaded by millions of users: `https://www.zdnet.com/article/fake-whatsapp-app-fooled-million-android-users-on-google-play-did-you-fall-for-it/`

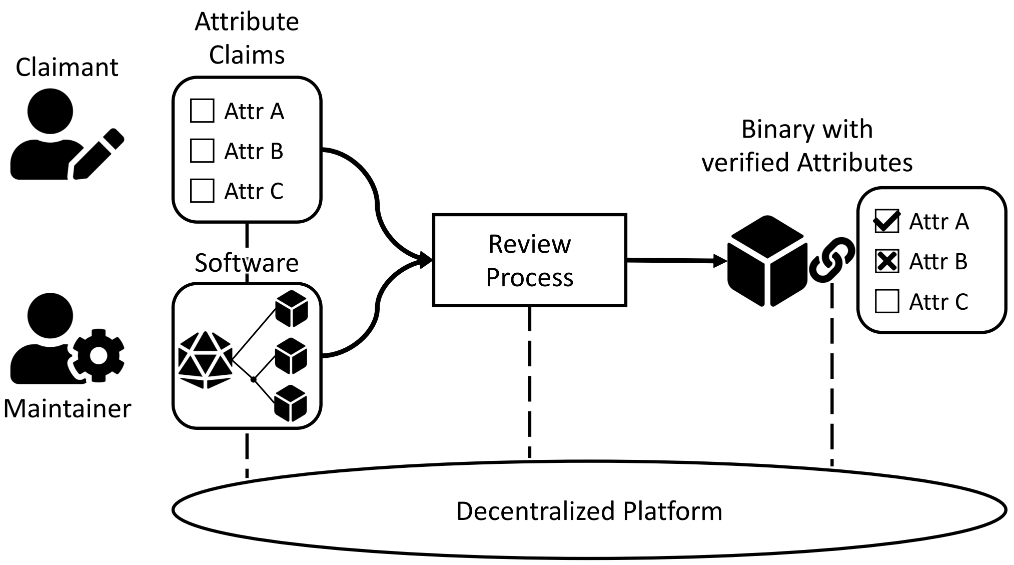


Figure 1.1.: The overall goal of our work is to support identity management, review, and attribute bindings for software binaries via decentralized platforms. While identity concepts for human actors (developers, claimants, reviewers) are well-known, we introduce explicit identities for software and its binaries to facilitate their review and make resulting attributes usable to end-users.

Furthermore, we endeavor to show the usefulness of software identities by conducting and documenting review processes on a distributed ledger to attach decentrally verified attributes concerning functionality or security characteristics to individual binaries of a software.

With decentralized consensus systems like Ethereum, a new area of research around their properties, capabilities, and limitations has opened up. Over the next decades, currently existing or future decentralized systems could rise to broader adoption by layperson users and become part of everyday digital life, much like the Internet in general is today. While it is outside the scope of this work to fully explore the long-term societal ramifications of such systems, we endeavor to contribute to answering overarching questions through the examination of a particular use case:

> *Can distributed ledger technologies enable the establishment, management, and use of software identities for the purpose of secure software distribution without a trusted third party?*

Once we realized a software identity management system on Ethereum, we aimed to enable and log an independent review of software releases on Ethereum as well. However, this revealed a new problem: The process of recording transactions on the Ethereum blockchain is not designed to facilitate a coordinated publication of multiple transactions. Yet, such a mechanism is necessary in order to enable reviewers to record their independent assessments of a software release and to make the disclosure process transparent afterwards. Thus, the second research question of this dissertation is as follows:

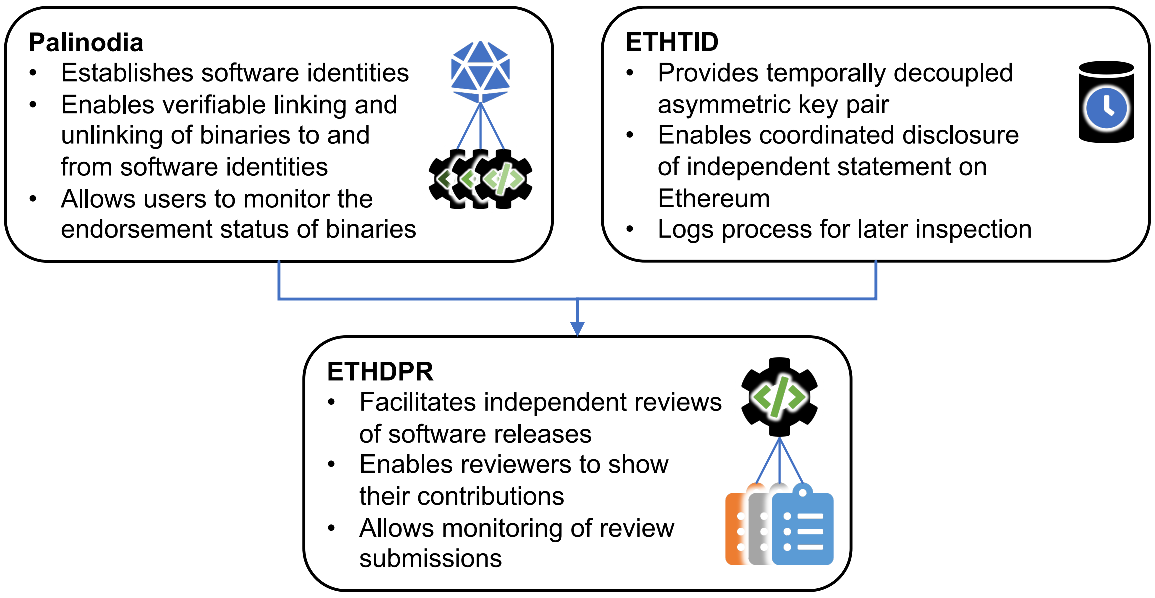

Palinodia
• Establishes software identities
• Enables verifiable linking and unlinking of binaries to and from software identities
• Allows users to monitor the endorsement status of binaries
ETHTID
• Provides temporally decoupled asymmetric key pair
• Enables coordinated disclosure of independent statement on Ethereum
• Logs process for later inspection
ETHDPR
• Facilitates independent reviews of software releases
• Enables reviewers to show their contributions
• Allows monitoring of review submissions

Figure 1.2.: Overview of the three main contributions of this dissertation. Palinodia enables software developers to establish and manage software identities. ETHTID facilitates a coordinated disclosure of independent statements on Ethereum through a temporally decoupled asymmetric key pair. ETHDPR combines Palinodia software identities and the ETHTID disclosure mechanism to perform and record reviews of software releases.

Can distributed ledger technologies support cryptographic protocols to achieve a coordinated disclosure of independent statements on a public ledger without a trusted third party?

1.1. Contributions

The primary contributions of this dissertation address the aforementioned research questions in three parts as depicted in Figure 1.2.

Palinodia We design, implement, and evaluate a software identity management system on Ethereum to empirically determine its feasibility and highlight potential improvements in the operations of Ethereum. We cover both the part of software developers who establish and manage software identities through smart contracts as well as the part of users who wish to obtain data from said contracts in order to validate downloaded binaries before and during their use. By enabling developers to both publish and revoke individual binaries of their software on-chain and by enabling users to reliably retrieve this data, we secure the software distribution process proactively against malicious actors and reactively against security-relevant bugs. We recognize the importance of access control to this use case and demonstrate that Ethereum smart contracts serve not only as referenceable data stores but also as reliable access control mechanisms.

ETHTID Independent of our concrete use case below, we define the coordinated disclosure problem and introduce one possible solution via *threshold information disclosure* by transferring well-established cryptographic mechanisms to Ethereum smart contracts. A council of configurable size is tasked via an Ethereum smart contract with the distributed generation of a temporally decoupled asymmetric key pair: An encryption key is generated in the present but the corresponding decryption key is held in a threshold-shared form and only recovered and published at a specified time in the future. Such a key pair can be used to publish independently generated statements in encrypted form that only become readable once the corresponding decryption key is published. We evaluate our implementation regarding scalability in the size of the council versus overall execution costs and expose fundamental limits and problems with such decentralized procedures. By carefully analyzing the cryptographic requirements of our approach and applying corresponding optimizations, we are able to reduce the execution costs of our implementation significantly compared to previous work. While we developed ETHTID to realize our next contribution, it might be of independent interest to other smart contract applications.

ETHDPR Lastly, we integrate Palinodia and ETHTID to enable a decentralized public review of software releases on Ethereum. By attaching a list of verification tasks to a software release in addition to an ETHTID instance for scheduling, a developer can initiate a time-bounded review process by volunteers. Each reviewer can perform the requested verification tasks to the best of their abilities and available tools and log their structured results in encrypted form on the Ethereum blockchain with references to both the software release in question and the ETHTID instance for eventual disclosure. Through a coordinated disclosure and the permanence of the resulting record, reviewers are placed under a mutual competitive pressure to perform their verification tasks accurately and thoroughly. Conversely, the logged results can be compared post-disclosure and thus serve as documentation for the performed verification independent of whether or not an unexpected discovery was made.

Methodology We tackle the aforementioned research questions by designing, implementing[4], and evaluating prototypical systems to demonstrate the viability of distributed ledgers for the use case of software identity management and coordinated software review. In the latter case, we overcome a functional limitation of our chosen platform in order to achieve our stated goals. In this way, we show the usefulness of decentralized platforms and how certain shortcomings can be overcome without sacrificing decentralization. By necessity, we also contribute to the development of a methodology to determine the practical feasibility of smart contract applications on Ethereum. In particular, as some of our implementations consist of smart contracts on-chain and clients running off-chain, we adopt a dual evaluation methodology for these execution environments. In this way,

[4] Our implementations are available as a git repository: `https://git.scc.kit.edu/dsn-projects/dissertations/dsim/`

we hope to make this emerging and evolving execution environment for decentralized applications more accessible to the wider scientific community.

Software Identity Concept As a necessary part of our work, we also contribute to the general concept of identity management for software and thereby enable further research and development on this subject, which we believe to be of increasing importance. While our work was enabled and to some extent informed by the emergence of decentralized platforms, we argue that the concept of software identities has existed implicitly before then. In this work, we provide an explicit concretization of what software identities require and enable.

Parts of the contributions of this dissertation have been published in the following previous works:

- O. Stengele and H. Hartenstein. Atomic information disclosure of off-chained computations using threshold encryption. In *International Workshop on Cryptocurrencies and Blockchain Technology (CBT), 2018*, pages 85–93, 2018. [83]
- O. Stengele, A. Baumeister, P. Birnstill, and H. Hartenstein. Access control for binary integrity protection using Ethereum. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 3–12, 2019. [85]
- O. Stengele, J. Droll, and H. Hartenstein. Practical trade-offs in integrity protection for binaries via Ethereum. In *Proceedings of the 21st International Middleware Conference Demos and Posters*, pages 9–10, 2020. [82]
- J. Schiffl, M. Grundmann, M. Leinweber, O. Stengele, S. Friebe, and B. Beckert. Towards correct smart contracts: A case study on formal verification of access control. In *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 125–130, 2021. [78]
- S. Friebe, O. Stengele, H. Hartenstein, and M. Zitterbart. Coupling smart contracts: A comparative case study. In *3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 137–144, 2021. [34]
- O. Stengele, M. Raiber, J. Müller-Quade, and H. Hartenstein. ETHTID: Deployable threshold information disclosure on Ethereum. In *Third International Conference on Blockchain Computing and Applications (BCCA)*, pages 127–134, 2021. [86, 87]
- O. Stengele, C. Westermeyer, and H. Hartenstein. Decentralized review and attestation of software attribute claims. In *IEEE Access*, vol. 10, pages 66694–66710, 2022. [84]

1.2. Thesis Outline

The remainder of this dissertation is structured as follows. In Chapter 2, we review fundamental concepts pertaining to Palinodia and ETHDPR. More specifically, we present a definition of software identities and give a historic review of implicit software identity management in various systems and contexts. We also elaborate on the components and construction of distributed ledgers at the example of Ethereum as the main environment for our implementations. In Chapter 3, we present Palinodia, our concept and implementation of a software identity management system on Ethereum. We also present a user client application to retrieve and use the information contained in Palinodia smart contracts. In Chapter 4, we introduce the coordinated disclosure problem independent of our later use case and present ETHTID, our concept and implementation of threshold information disclosure on Ethereum, as one possible approach to solve it. We provide separate fundamentals on number theory and threshold secret sharing in this chapter that underpin the construction of ETHTID. In Chapter 5, we integrate Palinodia and ETHTID into ETHDPR, a decentralized public review system for attribute claims of software binaries via Ethereum. We also provide a modularized generalization of our construction to aid in future improvements and extensions. We open Chapters 3 to 5 with dedicated problem statements before examining work related to their respective contributions, and provide system models and assumptions for the decentralized applications they cover. The evaluation of Palinodia, ETHTID, and ETHDPR in their respective chapters consists of a quantitative measurement of on-chain deployment and execution costs and off-chain effort as well as a qualitative semi-formal security evaluation. In Chapter 6, we discuss our results and their generalizability in relation to the aforementioned research questions. We then further discuss a Sybil problem that arises in Chapters 4 and 5 and potential ways to address it. Furthermore, we highlight two avenues to expand the concept of software identities and briefly explore what impact planned updates to Ethereum could have on the decentralized applications presented in this dissertation.

2. Fundamentals

Software has a remarkable position in modern society. In a sense, each software as a collection of digital objects, i.e. executable binaries, constitutes one logical tool in multiple distinct but related forms. Similar to physical tools, software is designed and crafted by one or more individuals but unlike physical tools, software is easily duplicated and disseminated around the globe. Consequently, it is no surprise that software of all kinds has attained critical importance on an individual, governmental, economic, and societal level.

In this chapter, we introduce concepts that are fundamental to our contributions. Specifically, we introduce the notion of software identity management and supplement an intuitive understanding with historic examples. At the example of Ethereum, we describe the components and constructions of distributed ledgers. Our contribution in Chapter 4 relies on threshold secret sharing, which we introduce separately in said chapter. Lastly, our contributions in Chapter 3 and Chapter 5 make use of off-chain data storage, which we introduce at the end of this chapter with a particular emphasis on the InterPlanetary File System, since we use it as a stand-in.

2.1. Software Identity Management

The term "software" encompasses a wide range of objects in both scope and complexity. From small command line programs and graphical desktop applications to entire software suites or the operating systems necessary to use them, to massive software projects present in modern day cars or planes. The concept of software identity we propose in this work deals with this scope by focusing on software distribution and inventory. As long as a software release has a well-defined binary representation that can be passed through a cryptographic hash function, the software identity management concept presented in this work is amenable to it. Dependencies on runtime environments like shared libraries could also be made explicit through separate software identities, a point we discuss further in Chapter 6.

In a very broad sense, the philosophical meaning of identity is the relation any object has only with itself, and conversely, what distinguishes objects from each other. Software, particularly when actively developed, presents interesting challenges to this concept both within and across versions. For example, the source code of a particular software version can be compiled into a plethora of distinct binaries due to compiler options or target platform, but functionally and semantically, all those binaries are still "the same software", just different expressions of it. Similarly, the same software can exist in multiple versions due to bug fixes or feature improvements, in which case such binaries are also functionally distinct. To deal with these issues, we regard software identities as meta constructs to which individual binaries are attached through authoritative sources, namely software creators.

The process of establishing a software identity, attaching and detaching binaries to and from it, and eventually decommissioning a software identity was already anticipated in the general identity management framework by the International Organization for Standardization and International Electrotechnical Commission (ISO/IEC) joint technical committee [44]. We reprint excerpts from the standard ISO/IEC 24760-1:2019[1] here for convenience, starting with the definition of an *entity*:

> **entity**
>
> item relevant for the purpose of operation of a *domain* that has recognizably distinct existence
>
> Note 1 to entry: An entity can have a physical or a logical embodiment.
>
> EXAMPLE A person, an organization, a device, a group of such items, a human subscriber to a telecom service, a SIM card, a passport, a network interface card, a software application, a service or a website.

Software takes the role of entity for which we establish both an overarching identity and partial identities in the form of attached binaries. An *identity* is defined as:

> **identity**
> partial identity
>
> set of *attributes* related to an *entity*
>
> [. . .]

In Chapter 5, we also enrich partial software identities with attributes, much like the standard ISO/IEC 24760-1:2019 allows:

> **identity management**
> **IDM**
>
> processes and policies involved in managing the lifecycle and value, type and optional metadata of *attributes* in *identities* known in a particular *domain*
>
> Note 1 to entry: In general identity management is involved in interactions between parties where *identity information* is processed.
>
> Note 2 to entry: Processes and policies in identity management support the functions of an *identity information authority* where applicable, in particular

[1] At the time of writing, this standard is publicly available: `https://standards.iso.org/ittf/PubliclyAvailableStandards/index.html`

to handle the interactions between an entity for which an identity is managed and the identity information authority.

In Chapter 3, we assign the role of identity information authorities to the developers and maintainers of a given software.

For the purpose of this work, we decompose the identity of a software into three aspects: It must provide a unique reference, ensure the integrity of attached binaries, and document the endorsement of binaries by the respective software creator. These aspects stand in a tight relation to each other: Without a unique reference to a specific software, both the protection of integrity and the endorsement of an authoritative party make little sense. Without integrity protection, both the reference and the endorsement could be applied to a maliciously altered version of a software. And lastly, without an explicit endorsement by the respective software creator, any statement about a particular software becomes worthless, since it no longer has to even be the same software. This last hypothetical also serves as a concise motivation for the importance of access control for software identity management.

Over the past fifty years of software creation and distribution, authenticity has been a prominent goal, although the perspective on the matter has shifted: In the early years, piracy was a significant problem for software creators, whereas consumers were willing to accept risks of using inauthentic and oftentimes modified versions of software, since personal computers had not risen to the level of importance they hold today. Nowadays, consumers have a vested interest in protecting their IT systems from harm or compromise. As a consequence, software users today pay more attention to the authenticity and legitimacy of software they use. Note that both of these cases relate back to the notion of software identity management as we address it in this work. However, despite its relevance, the management of software identities has gone mostly implicit as solutions arose incidentally in the way software distribution evolved over time.

While software was distributed on physical media, software identity management was straightforward, at least in theory: With CDs and DVDs being read-only, the integrity of the software in question was protected incidentally. Similarly, packaging and distribution chains through brick-and-mortar stores provided a clear endorsement by the authoritative software creator. Through inventory numbers, a generally low number of available software, and trademark law, unique referencing was also handled. However in practice, illegitimate versions of commercial software were created and distributed through the circumvention of copyright protection mechanisms. Such pirated copies were considered "the same software" by most users but were certainly not endorsed by the respective developers.

A more modern context in which software identities are established and managed incidentally or out of necessity are repositories. For example, the Comprehensive TEX Archive Network (CTAN)[2] is a repository of TeX packages, each of which have a unique, human

[2] `https://www.ctan.org/`

readable name as their identifier, like `cleveref`, `amsmath`, or `pgfplots`. In this case, a centralized repository manager ensures unique names or identifiers for software and an access control scheme ensures that only software and updates endorsed by the authoritative creator are published under the respective identifier. Integrity of downloaded software can be ensured through cryptographic means like digital signatures or securely distributed hash fingerprints. It is important to note that a repository manager represents a centralized trusted third party. One goal of our work is to establish software identity management akin to repositories without such a centralized party.

With the proliferation of Internet connectivity, especially broadband, throughout the 2000s, digital distribution platforms for software increased in popularity as well. Notable examples include Valve's Steam platform, the iOS and Mac App Store for mobile and desktop Apple devices respectively, Google's Play Store, and the Microsoft Store. Like repositories, these platforms also implement a form of software identity management out of necessity. Being more consumer-facing than repositories, digital software distribution platforms also include a convenient update mechanism for software. As with repositories, a platform maintainer represents a centralized party with the ability to block updates or completely remove any software. In recent years, there have been somewhat successful attempts at impersonating developers and then circulating bogus applications on these platforms for monetary gain[3].

Another context in which software identities play a significant role are trusted platform modules (TPMs) and more recently trusted execution environments (TEEs) like Intel's Software Guard Extensions (SGX)[4] or Keystone[5]. Broadly speaking, both TPMs and TEEs consist of hard- and software components to provide a secure environment for the execution of software. TPMs are used to establish a secure boot chain within a local system whereas TEEs are used particularly in remote and otherwise untrusted systems. Both technologies share the goal of achieving dependable and secure software functionality through cryptographic protocols executed in isolated and trusted hardware. For TPMs and in certain use cases of TEEs, a central aspect of their functionality is to verify that the right software is running. The ability to verify that a certain piece of software is running leaves open the external software identity management problem to determine *which* software is deemed correct by its authoritative creator. In this way, our work can complement and support certain use cases of TPMs and TEEs without relying on an additional trusted third party.

[3] In 2017, a fraudulent application disguised as an update to the widely-used messenger WhatsApp was published to Google's Play Store and downloaded by millions of users: `https://www.zdnet.com/article/fake-whatsapp-app-fooled-million-android-users-on-google-play-did-you-fall-for-it/`

[4] `https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html`

[5] `https://keystone-enclave.org/`

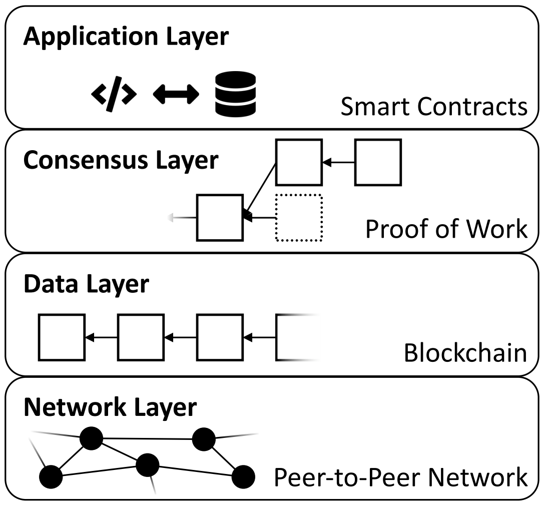


Figure 2.1.: DLT layer structure with corresponding components of Ethereum at the time of writing. Network layer forms the basis of communication; data layer defines the replicated data structure of the ledger; consensus layer resolves disagreements and ensures consistency of the ledger across replicas; application layer provides the desired overall functionality of the ledger.

2.2. Smart Contract-Capable Distributed Ledger Technologies at the Example of Ethereum

With the concept of software identity management as described above, we turn to smart contract-capable distributed ledger technologies as the primary means to realize a decentralized software identity management system. We focus our description on Ethereum, as it is the most prominent public and permissionless smart contract platform at the time of writing and thus the one we chose for our implementations. However, the general structure we present here has since established itself across multiple DLTs.

After Bitcoin [62] prominently demonstrated the viability of a permissionless consensus system with a distributed public ledger as its core data structure in 2009, Vitalik Buterin proposed Ethereum [17] as a more general platform for decentralized applications in 2013. Since then, Bitcoin and Ethereum have grown as both a technological and economic phenomenon with their respective cryptocurrencies reaching market caps in the billions and trillions of US dollars[6]. With the term “Ethereum”, we henceforth refer to the Ethereum mainnet both as a concept and as a concrete implementation, depending on context. This distinction is necessary as anyone can start a distributed ledger that is functionally identical to the Ethereum mainnet. Indeed, a primary evaluation tool for our work is to spin up an entirely local instance of an Ethereum blockchain to execute transactions.

Like many DLTs, Ethereum can be structured into four layers as depicted in Figure 2.1. All layers will be described in more detail in the following subsections, but due to their tight coupling and mutual dependencies, a brief introduction is provided for orientation.

[6] `https://coinmarketcap.com/historical/20211017/`

The main goal of Ethereum as a whole is to establish one public, deterministic “world computer”, also known as a *replicated state machine*. The state in question is replicated across nodes in an open peer-to-peer network where nodes can join and leave at any time. In order to handle the (re)joining of nodes, the state history is organized as a blockchain which is replicated across all nodes and provided to newly joined nodes on demand. Through its linear arrangement of blocks containing user requests in the form of transactions, a blockchain allows anyone to replay and verify the state evolution from the beginning, i.e. the genesis block, or from the last block they processed. A consensus mechanism ensures that the entire network agrees on exactly one state history by resolving conflicts that can emerge in the most recent state transitions. Due to the structure of a blockchain, consensus on the most recent blocks also implies consensus on the state history up to this point, thereby giving past transactions a high degree of permanence. Lastly, users interact with this world computer through transactions that can deploy stateful programs in the form of *smart contracts* or interact with them. Through the underlying layers, particularly the consensus mechanism, smart contracts can mediate processes between mutually distrustful parties as no single party can exert meaningful control over the ledger as a whole or a contract in particular beyond what it was programmed to do.

It is important to note that Ethereum remains in active development and, due to the nature of static documents, the description given here can only serve as a snapshot. Changes to the Ethereum protocol that impact its blockchain are grouped together into so called *hard forks*. In this work, we limit our view to the “London” hard fork that went live in August 2021. We note some changes to Ethereum that were applied since then in the following subsections and how they relate to our work.

2.2.1. Network Layer: Peer-to-Peer Network

The most fundamental layer of Ethereum is an open peer-to-peer network of nodes running an Ethereum client. At the time of writing, Ethernodes[7] and the node tracker of Etherscan[8] report around 3000 nodes on the main Ethereum peer-to-peer network, albeit with somewhat different country distributions. The primary task of this peer-to-peer network is to replicate, maintain, and make available the blockchain in addition to supporting both the consensus and application layer. To this end, nodes propagate transactions meant to alter the world state in some way which are then grouped together into new blocks in the consensus layer before being appended to the blockchain in the data layer. Since this is an open network and no individual node can be trusted, it is vitally important for the overall safety of Ethereum that nodes validate any data before accepting and forwarding it. The precise details of these validity checks are defined as part of the Ethereum protocol[9].

[7] `https://www.ethernodes.org`
[8] `https://etherscan.io/nodetracker`
[9] `https://ethereum.github.io/yellowpaper/paper.pdf`

In order to interact with the Ethereum peer-to-peer network, one has to run an Ethereum client and connect to peers on the network. Broadly speaking, there are two distinct modes that an Ethereum client can be run in: *full* or *light*. Full nodes form the peer-to-peer network as described above. They validate and propagate transactions and blocks while keeping a full copy of the entire blockchain. When a full node joins the network, it uses a variant of Kademlia [57] to discover and connect to other peers via TCP/UDP and announce itself. Once the new full node has established connections to other peers, it starts requesting blockchain data from them in order to establish and validate its own copy, a process called synchronization which can take several days and involves the transfer of a terabyte of data at the time of writing. By contrast, light clients save on resources by requesting, storing, and validating only the bare minimum of data, namely the block headers, which will be explained in the next subsection. Light clients connect to one or more full nodes in order to obtain block headers, send or observe transactions, and to obtain any piece of the world state. Similar to the interaction between full nodes, light clients also validate data they obtain from full nodes using the hash-linked structure of the blockchain. However, light clients do not contribute to the dissemination of blockchain data, they merely request, obtain, and validate data as instructed by applications or end-users.

2.2.2. Data Layer: Blockchain

As a logically centralized, evolving data structure, Ethereum's blockchain forms the basis for its consensus and application layer. While this subsection is focused on the construction of the blockchain itself, many of the properties required for Ethereum's proper functioning only emerge in conjunction with the consensus mechanism described in the next subsection. The decentralized applications we present in Chapters 3 to 5 make extensive use of the blockchain's structure, most notably the resulting ability to prove the existence, integrity, and inclusion of any part of Ethereum's past or current world state.

As the name suggests, a blockchain is comprised of individual blocks, which in turn consist of a block header and body. Block headers are sequentially numbered and chained together through a *cryptographic hash function*: Each header contains the hash of its immediate predecessor and thus transitively attests to the integrity of the header chain thus far. Additionally, each block header also contains the root hashes of three *Merkle Patricia Trie*s (MPT):

- Transactions root, containing all transactions that were executed in the consensus layer with this block
- Receipts root, containing receipts generated during the execution of those transactions
- State root, which is derived from the resulting world state with all account records, deployed smart contracts, and their respective state

Similar to Merkle trees [58], MPTs facilitate efficient proofs for the integrity and inclusion of any piece of data they contain. For any particular leaf of an MPT containing a piece of data, such a proof consists of the hash values of sibling nodes along the path from this leaf to the root of the tree. By first hashing the data in question and then successively appending or prepending the sibling hashes along the path and hashing the result, a verifier can recompute the original root hash and thus be convinced of the inclusion and integrity of said data. We make indirect use of these proofs in Chapters 3 to 5.

Another noteworthy part of block headers is the so called *logs bloom*, a Bloom filter [12] populated with data pertaining to *events*. During the execution of smart contract code, events can be emitted which can contain up to three indexed parameters and additional data. The address of the emitting contract as well as the indexed parameters are added to the logs bloom. This allows an Ethereum client to efficiently search its copy of the blockchain for relevant blocks and transactions given an event name or indexed parameters. As a probabilistic data structure, Bloom filters allow false positives, i.e. that a block meets a search query but does not actually contain a relevant transaction on closer inspection, but prevents false negatives, i.e. if a block does not meet a search query it does not contain a relevant transaction. We make use of this functionality in our smart contract applications to facilitate security relevant monitoring of events as well as to store and broadcast data within events.

Lastly, each block header also contains data related to the consensus mechanism covered in the next subsection.

2.2.3. Consensus Layer: Transactions, Fees, and Mining

With the general structure of the blockchain and the peer-to-peer network maintaining it covered, we now turn to actual interactions with the system in the form of transactions, their fee structure, and the process of appending new blocks to the blockchain.

Ethereum differentiates two kinds of accounts: externally-owned accounts and contract accounts, the latter of which will be covered as part of the application layer in the next subsection. Externally-owned accounts, as the name suggests, are defined by and controlled via an asymmetric key pair which is generated by users. Each externally-owned account is uniquely referenced by the last 20 bytes of the hash of its public key whereas transactions originating from such an account must be signed with the corresponding private key. The first time an externally-owned account appears as the recipient of a transaction, it becomes part of the world state and is henceforth logged in the state snapshot of each block.

Transactions are signed statements by holders of externally-owned accounts to change the world state in a particular way, for example by transferring Ether, deploying a smart contract, or calling a function of an already deployed contract. After constructing and signing a transaction, the account holder submits it to the peer-to-peer network where it is propagated if and only if it is valid according to the Ethereum protocol [90]. In order for a transaction to be executed and actually change the world state, it must be included in a block and become part of the canonical chain.

As mentioned previously, the blockchain as a data structure encodes *a* history of the world state machine of Ethereum, it is only through a consensus mechanism that it represents *exactly one* canonical and generally accepted history. At the time of writing, *mining* is the process by which extending this history is accomplished without a centralized party. Essentially, miners compete with each other to solve a cryptographic puzzle which is required by the protocol to form a new valid block. The difficulty of these puzzles is adjusted by the protocol such that, on average, a new block is found every twelve to fourteen seconds. In its header, each valid block contains both the current difficulty as well as a probabilistic proof of the exerted computational work, hence the name "Proof of Work" for this mechanism. In addition to decentralizing the privilege to extend the blockchain, mining also serves to establish consensus over the blockchain by resolving disputes. If two miners each propose a valid new block with the same predecessor block to the network at approximately the same time, then the blockchain is *forked*. All miners will then select either of the branches to extend and eventually one branch will become demonstrably longer, resolving the issue and reestablishing a linear, canonical history. Lastly, since each Proof of Work is cryptographically tied to its respective block, the consensus mechanism also serves to render the entire state history practically immutable: Any change to a previous block would require a new Proof of Work for this and all subsequent blocks and overtake the canonical chain in length before it would be considered part of the world state according to the Ethereum protocol. In addition to requiring more computing power than all honest miners combined, such an undertaking would hardly go unnoticed since it has to be performed in full view of the public.

It is important to note that, much like in Bitcoin before Ethereum, such a Proof of Work mechanism establishes a probabilistic consensus with no definitive finality. The fundamental assumption for both systems is that the majority of mining power is controlled by protocol-abiding, honest miners. Particularly the most recently added blocks can be rendered *stale* if a longer suffix to the chain appears on the network that excludes them. This is called a *chain reorg*. With each block appended to any particular block, the chances for a maliciously crafted or naturally occurring chain reorg drop exponentially. While a chain reorg is always possible for chain suffixes of any length in theory, in practice seven to 250 confirming blocks are considered sufficient to prevent a reorg. The choice for the number of confirming blocks is somewhat subjective and depends on the security demands of an application. With The Merge, which we explain shortly, this rule of thumb was made more explicit with *safe* and *finalized* stages for blocks being defined in-protocol and exposed to applications by Ethereum clients.

Since transaction in Ethereum can involve the execution of arbitrary smart contract code, it is vitally important to the system as a whole that no transaction can run forever. Ethereum accomplishes this through the *gas* mechanism, which will play an important role in our evaluations in Chapters 3 to 5. In addition to specifying what a transaction should do, a sender also specifies the *gasprice*, the conversion rate from Ether to gas, and the *gaslimit*, an upper limit to the amount of gas the transaction is allowed to consume. Each basic operation of the Ethereum Virtual Machine (EVM), the execution environment of smart contract code, has a fixed gas cost attached to it, as defined by the Ethereum yellow paper [90]. The execution of any transaction then either terminates with gas left over, which is reimbursed to the sender, or runs out of gas in which case no state changes are persisted but the transaction is recorded on-chain as failed. In either case, the consumed gas is collected by the miner who chose to include the transactions into their newly mined block.

Blocks in Ethereum are limited by the total amount of gas that their enclosed transactions can consume. Consequently, senders of transactions compete with each other over this limited space by setting the gasprice of their transactions depending on the urgency of their request, their financial means, and the current demand: The higher the gasprice, the more lucrative it is for a miner to include the transaction in a block, regardless of how much gas the transaction actually consumes. Conversely, lower gasprice transactions may take longer to be included in a block but are of course cheaper. With the aforementioned London hard fork, the protocol change EIP-1559[10] was deployed to the Ethereum main net which altered the fee mechanism described above significantly. However, these changes have no direct influence on the validity of our evaluation as they do not impact the gas costs per operation on the EVM. While the calculation for transaction fees and their distribution have changed, the slow upwards trend of the average gasprice has continued as interest in Ethereum grows. In Chapters 3 to 5, we use the exchange rates as reported by Etherscan[11] for July 1st 2022 as part of our cost evaluations.

Lastly, it is worth noting that, in September 2022, Ethereum successfully completed “The Merge” and transitioned from the Proof of Work consensus mechanism described above to a Proof of Stake mechanism where computational effort is replaced with staking of Ether itself to govern the privilege of appending new blocks to the chain. Our smart contract applications, much like Ethereum as a whole, rely on the existence, safety, and resilience of a consensus mechanism and not on its exact inner workings. In other words, the solutions we present and evaluate in Chapters 3 to 5 remain viable even after Ethereum’s transition to Proof of Stake.

2.2.4. Application Layer: Smart Contracts

First proposed by Nick Szabo in 1997 [88], smart contracts only became practically viable with the advent of decentralized public infrastructure in the form of blockchains. In 1999, Lawrence Lessig [50] also discussed the notion of “Code is Law” and its importance to shaping cyberspace. While the concept of mediating procedures between mutually distrustful parties through technological means like program code is interesting in itself, we will focus on the implementation of the concept within the context of Ethereum.

Ethereum smart contracts can be described as stateful programs with the additional ability to receive, hold, and distribute Ether. A simplified structural overview of three exemplary smart contracts is depicted in Figure 2.2. Each smart contract consists of program code, which determines both its behavior and an interface towards other accounts, and its current

[10] `https://eips.ethereum.org/EIPS/eip-1559`
[11] `https://etherscan.io/`

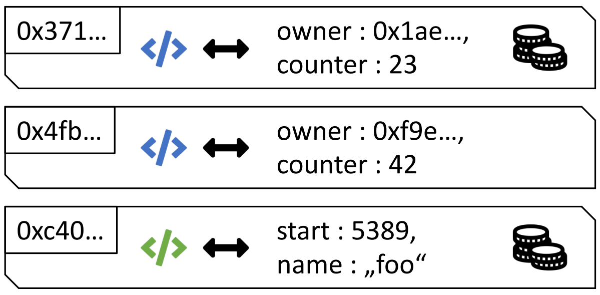


Figure 2.2.: Simplified structural overview of Ethereum smart contracts. Each contract has a unique address, shown abbreviated in their top left corner. The first and second contracts have identical code, signified by the colored </>, but distinct states. The first and third contract also hold Ether.

state, which can be accessed and modified during transactions according to the contract's programming. In the case of Ethereum, smart contracts are written in a high-level, object-oriented, statically typed language like Solidity[12] or Vyper[13] before being compiled to EVM bytecode and deployed to the blockchain through a transaction. During deployment, a contract account with a unique address is created that contains the executable bytecode in preparation for transactions directed to this account.

A contract's bytecode becomes part of the world state upon deployment. On the consensus layer, a deployment transaction for a smart contract establishes a system-wide, verifiable, and immutable relation between its newly assigned address, its codified behavior, as well as its past and current state. When calling a smart contract function, the sender of a transaction can know with certainty what the contract will do, pursuant to the contract's current state. While externally-owned accounts are ultimately controlled by humans through the use of an asymmetric key pair, contract accounts are entirely controlled by their associated EVM bytecode. However, Ethereum smart contracts are, at least currently, entirely reactionary, i.e. they can not issue transactions on their own but must instead be prompted into action by a transaction from an externally-owned account. Note that smart contracts can call other contracts during the execution of a transaction.

Next to its governing bytecode, each contract account also contains its own state which is tightly coupled to the overall functionality of the smart contract. Each contract's state can only be altered by its code and, likewise, the current state can affect the execution of transactions in accordance with the contract's code. This property results form the way transactions are executed in the consensus layer and persisted on the blockchain: Nodes in the peer-to-peer network holding a copy of the blockchain accept changes to the state of any contract only as the result of a valid transaction involving the execution of the corresponding contract bytecode. In this way, smart contracts are stateful programs on a blockchain with the added capability of receiving, holding, and redistributing Ether, depending on their programming.

[12] `https://docs.soliditylang.org/`
[13] `https://vyper.readthedocs.io/`

Much like the blockchain itself, smart contracts are public in more ways than one: Both the bytecode and its state are available for anyone to examine; but more crucially, anyone can interact with a smart contract through a transaction unless codified otherwise. As a consequence, an important aspect of smart contract design and implementation revolves around access control, which will be covered in more detail in Chapters 3 and 5.

In this work, smart contracts are not only the primary tool we use to establish, manage, and augment software identities, they are also an instructive implementation of software identity themselves. As mentioned previously, each Ethereum smart contract is assigned a unique address during deployment. For example, in Figure 2.2, the first and second contract have exactly the same bytecode but distinct addresses and states. Consequently, a contract address uniquely identifies a particular *instance* of a smart contract rather than the contract as an abstract object. However, it is exactly this overarching object which is distinct from but still related to any particular instance of a software that we endeavor to make concrete and manage with our work.

2.2.5. Distributed Ledgers as Logical Clocks

Distributed ledgers, by construction and definition, are ever-changing systems. New transactions are constantly being submitted to peer-to-peer networks, grouped into blocks and appended to the respective blockchain, and the corresponding world state changes accordingly. For the proof-of-concept constructions we present in Chapters 3 to 5, it is worth emphasizing and concretizing the notion of logical time and order a distributed ledger like Ethereum provides.

On the data layer that we describe in Subsection 2.2.2, a blockchain like Ethereum's consists of sequentially numbered blocks. This sequential numbering establishes a total order for blocks and the transactions they contain[14]. The block number of the most recent block is also called Ethereum's *block height*. Crucially, the sequence number of a block is accessible in the EVM during the execution of contained transactions via a Solidity constant. This way, the outcome of transactions can depend on whether a given block height has been reached or not. It is important to note, particularly for our approach in Chapter 4, that such a an application layer deadline mechanism cannot be used to prevent the dissemination of information contained in a transaction. Even if a transaction is aborted due to a contract-defined block height not being reached yet, any information contained in such a transaction is publicly logged on the data layer as part of its execution on the consensus layer.

In Chapter 4, we model Ethereum as a synchronous public broadcast channel. In doing so, we abstract away the uncertainty involved with broadcasting messages on Ethereum. Transactions are included in blocks at the discretion of their respective miners, as we explain in Subsection 2.2.3, and due to current demand, the time between transaction submission and its inclusion in a block is neither predictable nor bounded. Depending on

[14] The transactions within each block are also totally ordered by the proposing miner.

the chosen gasprice in relation to other pending transactions, this inclusion can happen sooner, later, or not at all. Especially in conjunction with contract-enforced deadlines as described above, it is important to choose the corresponding block numbers and intervals with appropriate leeway.

2.3. Decentralized Off-Chain Storage Systems

Similar to companies like Amazon renting out spare storage resources, so too have decentralized markets for storage space emerged based on distributed ledgers. At a high level, decentralized storage systems can be decomposed into two layers: an incentive layer and a storage layer. While we only rely on the latter in this work, it is important to have a basic understanding of both layers to recognize the trade-offs involved in their use. Two systems that exemplify this two-layer structure are Filecoin[15] and Crust[16], which have both established their own distributed ledger as their incentive layer and use the InterPlanetary File System (IPFS)[17] for file management.

On the incentive layer, contracts between storage providers and customers are logged and enforced. In such contracts, the terms for the storage of files are set, like duration, number of replications, costs, and obligations for access, i.e. whether a party has to reimburse the storage provider to download files or whether the customer who uploaded said files has prepaid for a certain number of downloads. In order to receive payment for their service, storage providers must prove that they actually hold the files they agreed to store, the details of such proofs are particular to each system and not relevant for this work.

The file management layer, meanwhile, enables nodes to locate other nodes currently hosting files of interest to them, obtain files from nodes, and provide files to other nodes that they are interested in. We simplify a more complex stack of protocols and systems that make up IPFS for the sake of presentation. For a more detailed characterization of IPFS, we refer to the work of Henningsen *et al.* [43].

Files on IPFS are identified based on their content, rather than their storage location, as the latter can change constantly. A content identifier (CID) based on cryptographic hash functions provides both a unique reference to a file on IPFS and allows anyone to verify its integrity once obtained. Using a Kademlia distributed hash table [57], nodes currently hosting files announce themselves with the corresponding CIDs. To obtain a file via IPFS given its CID, a client queries this hash table to locate nodes hosting said file. After obtaining a file from a hosting node or nodes, an IPFS client verifies its integrity by recomputing and comparing its CID to the one a user or application requested. Depending on which hash function is used, CIDs in IPFS can be of varying length, as described by the IPFS documentation[18]:

[15] `https://filecoin.io/`
[16] `https://www.crust.network/`
[17] `https://ipfs.io/`
[18] `https://docs.ipfs.io/concepts/content-addressing/`

- CIDv0: 46 characters
- CIDv1 with sha256: 60 characters
- CIDv1 with blake2b-256: 63 characters
- CIDv1 with sha3-512: 111 characters

Note that IPFS on its own is not a storage system in the strictest sense: Users who wish to distribute a file through it must remain online and available until said file has been requested and disseminated sufficiently many times to ensure its continued availability even when the original uploader goes offline. Nevertheless, IPFS suffices for the systems we describe in Chapters 3 and 5 to log and time-stamp integrity-protecting references to files too big to store on distributed ledgers directly. For the purpose of our evaluations, the lengths of these references matter rather than the way in which they are resolved to obtain actual files. As such, other decentralized off-chain storage systems can be used as well, the ramifications of which we discuss in Section 5.7.

3. Palinodia: Software Identity Management on Ethereum

The content presented in this chapter has been published previously in the paper "Access control for binary integrity protection using Ethereum" by Stengele, Baumeister, Birnstill, and Hartenstein and presented at the 24th ACM Symposium on Access Control Models and Technologies [85], as well as the demonstration titled "Practical trade-offs in integrity protection for binaries via Ethereum" by Stengele, Droll, and Hartenstein given at the 21st international Middleware Conference [82][1].

With the proliferation of distributed ledger technologies, particularly blockchains, over the past decade, the notion of *self-sovereign identities* [61, 31] also garnered attention. Broadly speaking, self-sovereign identities are established, managed, and fully owned by the user they represent, independent of any identity provider or service. Public blockchains are particularly well suited to function as a platform for self-sovereign identities due to their continued existence not depending on any single authority and their core properties as described in Chapter 2. In this chapter, we examine the case of self-sovereign identities on public blockchains at the example of Ethereum not for users but for software and associated binaries.

3.1. Problem Statement

The overarching problem we tackle with our software identity management concept and implementation is to secure the distribution process of software binaries against malicious actors and augment it with a robust revocation mechanism, all without relying on a trusted third party. We consider a software distribution secure if a user can verify the integrity, authenticity, and creator endorsement of a downloaded binary. A binary has integrity if it was not modified by an unauthorized party; it is authentic if and only if it was created by an authoritative source; and it is endorsed if it has not been revoked by its creator.

Based on the above problem statement, it follows immediately that access control is of vital importance to our solution. Only by ensuring that no party other than the corresponding software developer can make changes to a given software identity does it become useful. It is similarly important to protect users, or rather user clients, from being convinced

[1] A video recording of this demonstration is available on YouTube: `https://www.youtube.com/watch?v=0AVd5sS2mCc`

of false information and tricked into using maliciously altered, inauthentic, or revoked binaries. We approach this problem by designing and implementing a software identity management concept that takes advantage of inherent properties of distributed ledgers, particularly Ethereum, as described in Chapter 2. In particular, software developers should be able to establish and manage identities for their software by attaching and revoking binaries through reliably access-controlled means. Meanwhile, users should be able to reliably obtain and verify information pertaining to the identities of software they intend to use in order to check the integrity, authenticity, and creator-endorsement of binaries they downloaded. The software identity management system we present here also serves as the foundation for the decentralized review and attestation of software claims in Chapter 5.

The remaining chapter is structured as follows. We first present and discuss related work in Section 3.2 before describing our system model including attacker capabilities and trust assumptions in Section 3.3. With Section 3.4, we present Palinodia, our decentralized software identity management system on Ethereum, consisting of on-chain smart contracts and an off-chain user client. We evaluate the efficiency and performance of Palinodia in Section 3.5 and show semi-formally how Palinodia's security properties reduce to the security properties provided by Ethereum. To end the chapter, we discuss our findings and highlight limitations and avenues for future work in Section 3.6.

3.2. Related Work

It is worth examining prior and concurrent works in the intersection of blockchains and software identity management.

Contour by Al-Bassam and Meiklejohn [7] is one of the earlier approaches utilizing blockchains to augment software distribution. The core of their concept is for a trusted software distributor to construct a Merkle tree [58] from several binaries, persist the root hash of this tree on the Bitcoin blockchain, and attach metadata with a proof of inclusion to each binary. Users of such binaries can then obtain the root hash, whose immutability and availability is ensured by the Bitcoin blockchain, and verify the proof of inclusion to convince themselves of the integrity of the binary they obtained. The goal of Al-Bassam and Meiklejohn was not to establish software identities but rather to ensure "software transparency", and indeed there are crucial and instructive differences between their work and ours to point out. First, a trusted software distributor is necessary to collect hash fingerprints of binaries to construct and publish a Merkle tree root hash. Consequently, the respective software developers have to trust this distributor to not also include maliciously altered versions of their binaries in said root hash, for example. Similarly, neither root hashes nor the binary hash fingerprints they contain can be revoked. Lastly, since the hash fingerprint of each binary must be generated before the root hash, the integrity of metadata attached to binaries can not be protected. This in turn necessitates an additional mechanism that enables users to distinguish between authentic and counterfeit root hashes on the Bitcoin blockchain.

An approach more comprehensive than Contour is Chainiac by Nikitin *et al.* [66]. The authors present a decentralized framework to establish update logs for software with a particular focus on source-to-binary correspondence. While not directly managed by developers, these update logs can be understood as a form of identity representation for software, as they also verifiably link together past and future releases of said software. However, unlike Palinodia, Chainiac offers no mechanism to revoke a particular binary once it is added to its update log and instead focuses on minimizing causes for such revocations through thorough reviews and verifiable build processes. In Chapter 5, we extend Palinodia with similar functionalities. Lastly, Chainiac does not rely on any existing blockchain or distributed ledger, instead opting to establish "skipchains" for each software, which can be aggregated for user convenience. In Section 3.6, we elaborate on the implications of building Palinodia on top of Ethereum.

Contemporary to our work, Guarnizo *et al.* present SmartWitness [37]. Similar to Palinodia, SmartWitness also employs smart contracts to establish identities for software and attach released binaries to it. However, Guarnizo *et al.* chose to integrate certificate authorities into their system design as creators and stewards of SmartWitness contract instances. While software developers still own their SmartWitness instances and are authorized to attach new binaries to it, the issuing certificate authority retains both the obligation to renew SmartWitness instances as well as the ability to revoke SmartWitness instances entirely. Consequently, SmartWitness instances are not self-sovereign software identities. Guarnizo *et al.* also installed the aforementioned certificate authorities as gatekeepers for "security providers". These security providers are tasked with performing analysis of binaries logged in SmartWitness instances and are authorized by certificate authorities to record their findings as a numerical score within said instances. Unlike the extension to Palinodia we present in Chapter 5, SmartWitness does not employ a mechanism to ensure the independence of security assessments. SmartWitness also handles metadata attached to binaries and the verification of binaries by users differently than Palinodia, a point we explore further in Subsection 3.4.2.

3.3. System Model

It is instructive to first give a more detailed software identity model before describing the roles and assets involved in Palinodia as well as our attacker and trust model.

For Palinodia, we model software identities as trees of height two, as depicted in Figure 3.1. The root of each tree, and the cornerstone of each software identity, is a *root software identity*. Attached to each root software identity are one or more *intermediary software identities* that represent, for example, more concrete facets of a software identity grouped by target operating system or device type. Lastly, each *binary* of a given software is attached to at least one intermediary software identity, thereby attaching it indirectly to its root software identity.

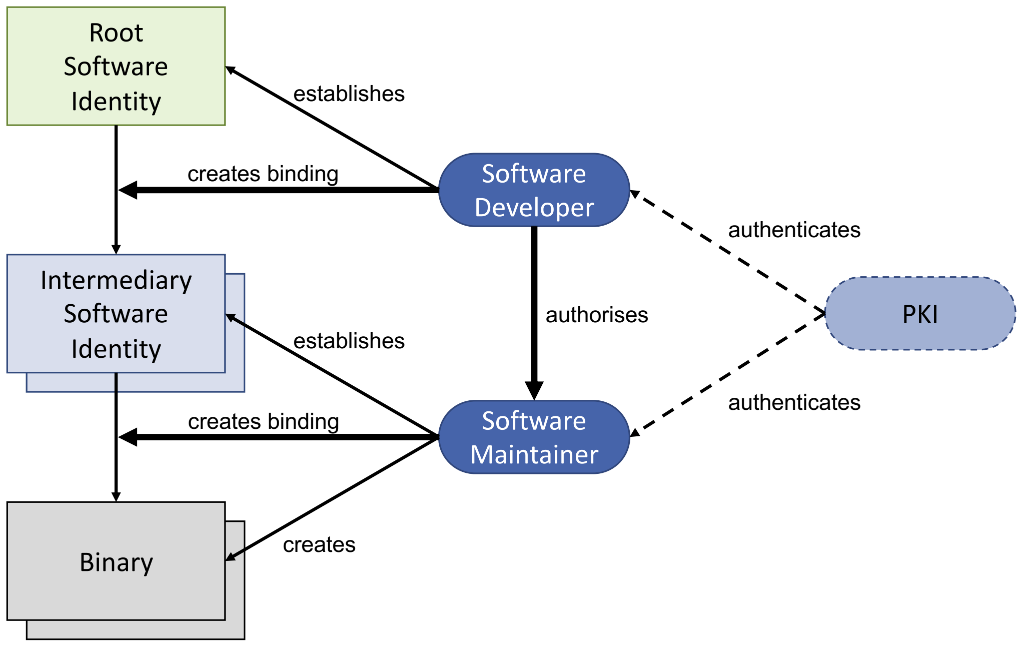


Figure 3.1.: Software identity hierarchy on the left and roles on the right with their capabilities displayed through annotated arrows. The authentication by a conventional PKI is optional.

3.3.1. Roles

We now introduce the roles depicted in Figure 3.1 in more detail. Rather than shouldering all privileges and responsibilities on a single authority, we opted to define two roles to demonstrate delegation and more fine-grained access control over a software identity. However, both roles can be occupied by the same entity.

A *software developer* takes responsibility for a software by creating it in the first place and refining its code base over time. They establish and take ownership of a root software identity. A *software maintainer* is authorized by a developer to create and distribute executable binaries from the code base of their software. Similar to developers, maintainers establish and take ownership of intermediary software identities to which they attach newly released binaries and from which they can also remove binaries that they consider no longer fit for use. Likewise, developers attach and detach intermediary software identities to their root software identity as needed, thereby authorizing or deauthorizing the corresponding maintainers to release binaries of a given software. By detaching an intermediary software identity, a developer effectively revokes all binaries associated with it in bulk.

In addition to the two roles explained above that are actively involved in establishing and managing software identities, we also define three roles to illustrate various use cases for software identities and to define an attacker and trust model in the next section. While we examine the distribution of metadata pertaining to software identities in great detail, we abstract the distribution of actual binaries to *software distribution platforms* (SDPs) which are controlled by *SDP owners* and have unique identifiers. For the purpose of this work, we use software distribution platforms as an umbrella term for repositories, app stores, software marketplaces, download centers, and other means of distributing software to end users that are managed and operated by a centralized authority. *Users* wish to obtain and use authentic software releases as well as to stay informed about new releases and revocations of software they use. Lastly, we define *auditors* as a special kind of user who obtains and verifies software releases not only for their own use but to verify the congruence between binaries attached to software identities and those available for download on SDPs.

3.3.2. Attacker and Trust Model

In order to evaluate the functionality of our proposed software identity management concept and highlight noteworthy features, we borrow the attacker model by Cappos *et al.* [19] and modify it for the context of blockchain applications.

Generally, the goal of an attacker is to trick a user into installing and running manipulated or vulnerable software to facilitate further compromise of the user's system. To achieve this goal, an attacker can modify existing software binaries or create entirely new binaries, either of which they can distribute via SDPs. Similarly, SDP owners can act maliciously and attempt to compromise users via their software distribution platform. Regarding the underlying blockchain of our concept, an attacker can deploy their own smart contracts and interact with existing smart contracts, like any other user. We also enable an attacker to occasionally obtain all but one particular private key from software developers and maintainers and use it to issue valid transactions in their name. Since we build on top of a public blockchain, an attacker can also join its consensus mechanism as a miner. However, we must assume that an attacker cannot obtain the majority of mining power as that would fundamentally break the blockchain system as a whole. An attacker is also unable to prevent read and write access to the underlying blockchain indefinitely. Similarly, an attacker can not break cryptographic primitives like signatures or hash functions.

Based on these goals and capabilities, we describe the following attacks based on Cappos *et al.* [19]:

- **Arbitrary binary**: An attacker tries to convince a user to install an untrustworthy binary.
- **Replay attack**: An attacker attempts to present outdated versions of legitimate binaries as current to users in order to exploit vulnerabilities that have since been discovered in them.
- **Freeze Attack**: Similarly to replay attacks, an attacker tries to prevent users from obtaining current information on a given software, particularly revocations.

Opposing an attacker are trust relations between users, software developers and maintainers, and the underlying blockchain system. Users trust software developers and maintainers to not intentionally include malicious functionality in their binaries and that, in the event of vulnerabilities being discovered, they revoke their endorsement of binaries

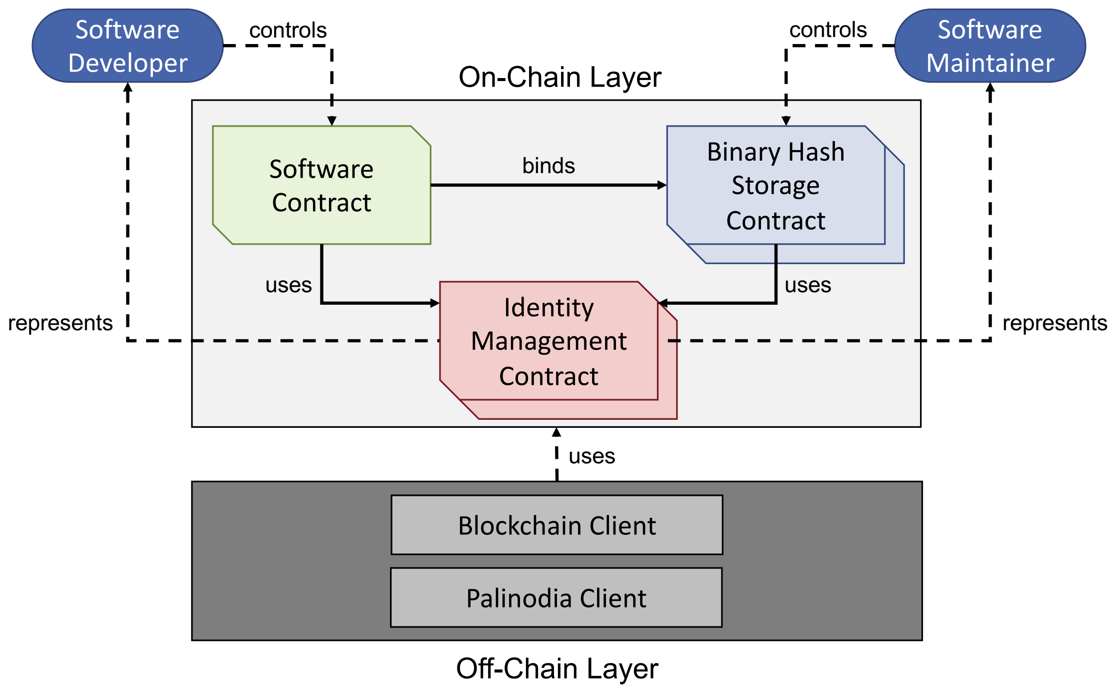


Figure 3.2.: On- and off-chain components of Palinodia in relation to the two active roles.

in a timely manner. Users similarly trust developers to manage their software identity with due diligence, particularly when transferring ownership of it to a new developer, or authorizing and deauthorizing maintainers to manage intermediary identities of their software. Lastly, users, developers, and maintainers trust peers running the underlying blockchain system to follow and enforce the system's protocol not individually, but as a collective.

3.4. Palinodia

We now present Palinodia, a software identity management system on Ethereum. As depicted in Figure 3.2, Palinodia is comprised of an on-chain layer in the form of smart contract instances and an off-chain layer consisting of an unmodified Ethereum client and a custom Palinodia client. On-chain, smart contract instances both represent root and intermediary software identities and also function as an access control enforcement mechanism over said identities. The off-chain components, meanwhile, enable users to perform verification of downloaded binaries and continuously monitor the Ethereum blockchain for new releases and revocations of previously released binaries.

3.4.1. On-Chain: Smart Contracts

We begin by describing common features of Palinodia's smart contracts before describing each contract type in more detail.

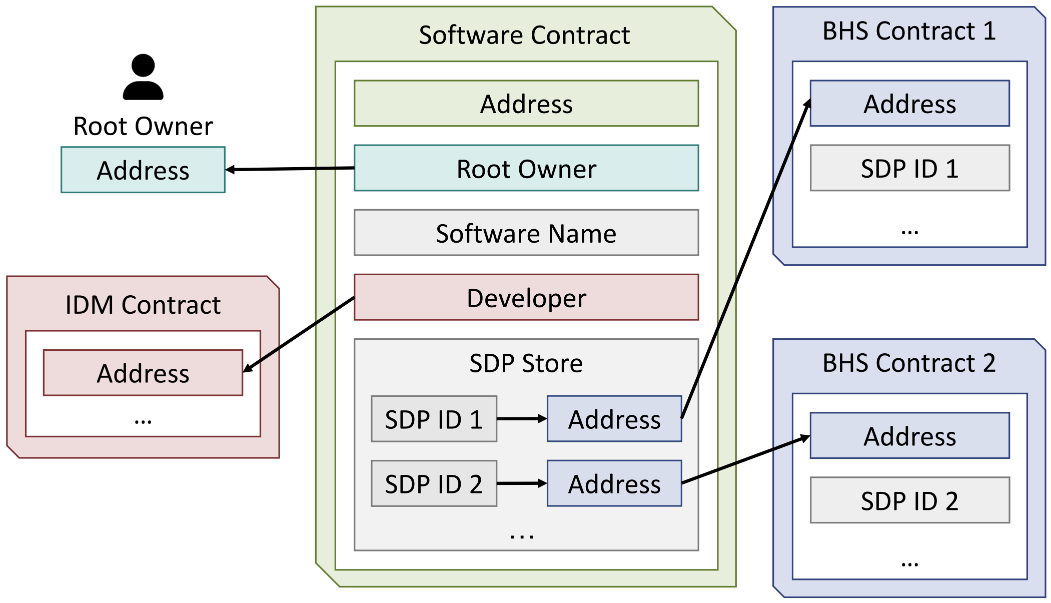


Figure 3.3.: Overview and relation of a Software contract to other Palinodia contracts.

Fundamentally, each contract type in Palinodia acts as an access-controlled key-value registry of contract instance addresses or hash fingerprints of binaries, thereby representing a software identity hierarchy as depicted in Figure 3.1. While access control during normal operations is handled through Identity Management contract instances, which we describe below, ownership of each contract instance and the highest administrative privileges are tied to a *root owner* address stored within each contract type. The private keys of root owner addresses are meant to be stored securely, i.e. in a physical safe or an air-gapped system, and only be used during contract deployment, when transferring ownership of contract instances, or in cases of emergency. This precaution allows a software developer or maintainer to regain control of their contract instances should one of their private keys in active use be lost or compromised.

Another common feature of all Palinodia contracts is the use of Ethereum events as introduced in Subsection 2.2.2 to facilitate monitoring of important events, such as changes in instance ownership, or the publication and revocation of binary hashes. Please recall the functionality of indexed event parameters to fine-tune search and monitoring queries.

3.4.1.1. Software Contract

The basis for each software identity in Palinodia is a Software (SW) contract instance deployed and managed by a software maintainer to represent a root software identity as depicted in Figure 3.1. Table 3.1 gives an overview of its interface. In addition to storing human-readable data about a software, like its name, the primary purpose of a SW contract instance is to register and deregister Binary Hash Storage (BHS) contract instances and store their addresses keyed by their respective SDP IDs as depicted in Figure 3.3. By adding a BHS contract instance to their SW contract instance via a `registerBHSContract`

Table 3.1.: Interface of SW contract. Simple calls to retrieve values of attributes are omitted [85]. Indexed event parameters are underlined.

Name	Arguments	Functionality	Allowed Caller
Transactions			
`changeRootOwner`	$\mathrm{ROwner}_{\mathrm{addr}}$ (new)	Replaces the stored root owner address.	*Root Owner* (cur.)
`setDeveloper`	$\mathrm{IDM}_{\mathrm{addr}}$	Replaces the stored developer address.	*Root Owner*
`setSoftwareName`	*name*	Sets the variable `software name`.	*Developer*
`registerBHSContract`	$\mathrm{BHS}_{\mathrm{addr}}$	Registers a BHS contract instance by storing the submitted $\mathrm{BHS}_{\mathrm{addr}}$. The SDP ID used as storage key is obtained from the BHS contract instance.	*Developer*
`deregisterBHSContract`	SDP ID	Deregisters the BHS contract instance stored under *SDP ID*.	*Developer*
`updateSDP_ID`	SDP ID (old), SDP ID (new)	Changes the storage key of a BHS contract instance.	*corr. BHS contract instance*
Calls			
`getBHSContract`	SDP ID	Returns the $\mathrm{BHS}_{\mathrm{addr}}$ stored under *SDP ID*. Returns 0, if no such entry exists.	*All*
Events			Emitting Function
`ROwnerChange`	$\mathrm{ROwner}_{\mathrm{addr}}$ (old & new)	Root owner change.	`changeRootOwner`
`DevChange`	$\mathrm{IDM}^{\mathrm{Dev}}_{\mathrm{addr}}$ (old & new)	Developer change.	`setDeveloper`
`DeregisterBHS`	<u>SDP ID</u>	Deregistration of BHS contract instance.	`deregisterBHSContract`

transaction, a developer can authorize the maintainer owning this BHS contract instance to publish binaries of their software as described in the next section. During this registration process, the address of the SW contract instance is stored within the BHS contract instance to facilitate binary verification, which we describe in more detail in Subsection 3.4.2. It is important to note at this point that we treat the globally unique address of each SW contract instance as the identifier of a software identity since it remains constant even if a developer transfers ownership of a SW contract instance, and thus the software identity.

Next to changes in ownership, the `deregisterBHS` event is noteworthy as it essentially signifies a revocation of all binaries registered in the BHS contract instance previously referenced under the attached SDP ID.

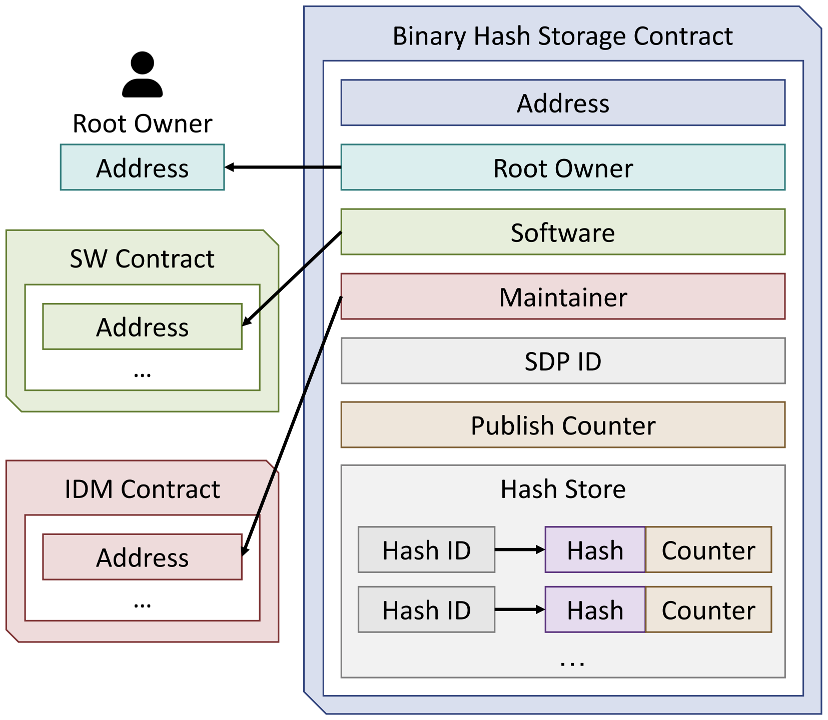


Figure 3.4.: Overview and relation of a Binary Hash Storage contract to other Palinodia contracts.

3.4.1.2. Binary Hash Storage Contract

Similar to SW contracts, Binary Hash Storage (BHS) contract instances represent intermediary software identities as depicted in Figure 3.1 and are deployed and managed by a maintainer. Table 3.2 provides an overview of their interface. As the name implies, BHS contract instances function as an access-controlled registry for hash fingerprints of binaries to both facilitate a verification of their integrity and to signify their maintainer's endorsement. Each hash fingerprint is keyed by a *Hash ID*, an identifier chosen by a maintainer as part of a `publishHash` transaction, which also emits a `publication` event on success. For Palinodia, the main purpose of `publication` events is for users to be notified of new releases of software they use. As such, we only defined the publication counter, which we explain in more detail below, as the only indexed parameter. For the extension of Palinodia we present in Chapter 5, this event takes on an additional role and the indexing of its parameters is changed accordingly. In addition to checking that the sender of such a transaction is authorized to add a new hash, the BHS contract code also ensures that the chosen Hash ID has not been used before, thereby making Hash IDs unique within each BHS contract instance. Consequently, a Hash ID together with a corresponding BHS contract instance address serves to uniquely identify a particular binary of a software that can also be verifiably traced back to its root software identity. This feature is crucially important for Chapter 5. To revoke the endorsement of a binary and prevent future verification, a maintainer issues a `revokeHash` transaction with the Hash ID in question as the only parameter. After ensuring proper authorization of the sending address, as described above, the corresponding hash is set to 0 to mark it as revoked and a `revocation` event

Table 3.2.: Interface of BHS contract. Simple calls to retrieve values of attributes are omitted [85]. Indexed event parameters are underlined.

Name	Arguments	Functionality	Allowed Caller
Transactions			
`changeRootOwner`	$\text{ROwner}_{\text{addr}}$ (new)	Replaces the stored root owner address.	*Root Owner* (cur.)
`setMaintainer`	IDM_{addr}	Replaces the stored maintainer address.	*Root Owner*
`registerSWContract`	-	Completes the binding to an SW contract instance.	*corr. SW contract instance*
`setSDP_ID`	SDP ID	Changes the stored SDP ID and calls the stored SW contract instance to update its corresponding storage key.	*Maintainer*
`publishHash`	HashID, Hash	Stores *Hash* under *HashID*.	*Maintainer*
`revokeHash`	HashID	Revokes the hash of *HashID*.	*Maintainer*
Calls			
`getBinaryStatement`	HashID	Returns the *binary statement* consisting of (Hash, Counter) stored under *HashID*. Returns 0, if no entry exists.	*All*
Events			Emitting Function
`ROwnerChange`	$\text{ROwner}_{\text{addr}}$ (old & new)	Root owner change.	`changeRootOwner`
`MaintChange`	$\text{IDM}^{\text{Maint}}_{\text{addr}}$ (old & new)	Maintainer change.	`setMaintainer`
`Publication`	$\underline{\text{cntr}}$, HashID, Hash	Publication of hash.	`publishHash`
`Revocation`	$\underline{\text{HashID}}$	Revocation of hash.	`revokeHash`

with the corresponding Hash ID as an indexed parameter is emitted. In Solidity, reading an unassigned variable returns a default value rather than an error. To distinguish an unused Hash ID from a revoked one, each BHS contract instance keeps a *publish counter* and attaches its current value to each hash upon its registration before incrementing its value. An unused Hash ID thus returns both a hash and a publish counter of 0, whereas a revoked Hash ID returns a non-zero publish counter.

3.4.1.3. Identity Management Contract

The last contract type in Palinodia are Identity Management (IDM) contracts. Their interface and functionality is presented in Table 3.3. IDM contract instances allow developers and maintainers to use multiple Ethereum addresses for any particular Palinodia instance and rotate them for convenience or security reasons in one contract instance rather than several. This functionality is particularly useful if the role of developer or maintainer is actually held by more than one person, each with their own set of addresses. During

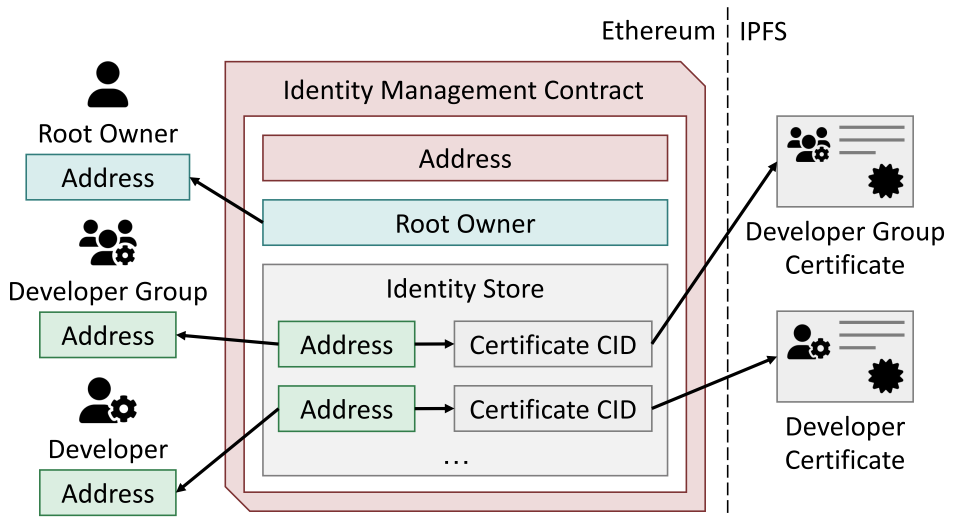


Figure 3.5.: Overview of an Identity Management contract and its relation to actors and additional credentials stored on IPFS.

Table 3.3.: Interface of IDM contract. Simple calls to retrieve values of attributes are omitted [85]. "*Identity store*" as allowed caller signifies that any identity currently in the identity store is authorized.

Name	Arguments	Functionality	Allowed Caller
Transactions			
`changeRootOwner`	ROwner$_{addr}$ (new), Cert. CID	Replaces the stored root owner address.	*Root Owner* (cur.)
`resetIdentitySet`	-	Resets `identity store` by removing all stored identities.	*Root Owner*
`addIdentity`	Identity$_{addr}$	Adds address of *identity* to `identity store`.	*identity store*
`removeIdentity`	Identity$_{addr}$	Removes address of *identity* from `identity store`.	*identity store*
`changeCertificateCID`	Cert. CID (new)	Replace the Cert. CID of the calling identity.	*identity store*
Calls			
`checkIdentity`	Identity	Returns `true` if *identity* is contained in `identity store`, returns `false` otherwise.	*All*
`getIdentityCertCID`	Identity	Returns the stored Cert. CID of *identity.*	*All*
Events			Emitting Function
`ROwnerChange`	ROwner$_{addr}$ (old & new)	Root owner change	`changeRootOwner`
`Reset`	ROwner$_{addr}$	Reset of `identity store`.	`resetIdentitySet`

critical transactions in SW or BHS contract instances, the authorization of the sending address is checked via the `checkIdentity` call to the IDM contract instance linked within the respective SW or BHS contract instance. Additionally, IDM contract instances allow developers or maintainers to attach further credentials, like a certificate, to their stored Ethereum address via an IPFS CID. It is important to note that IDM contracts function as a stand-in for more comprehensive user identity management systems on Ethereum, like DecentID [33] or other blockchain-based self-sovereign identity systems [74]. We examined approaches for such couplings at the example of Palinodia and DecentID in collaboration with Friebe and Zitterbart [34].

3.4.2. Off-Chain: Palinodia Client

With the on-chain components of Palinodia introduced in the previous section, we now describe the remaining off-chain components and how they facilitate verification of binaries and monitoring for notable events like revocations.

We start with Palinodia-specific additions to binaries and explain the essential operations of Palinodia in the order of a typical binary life cycle: from an initial release and update to its potential revocation. Bundled together with an executable binary is a manifest of Palinodia-specific metadata, most importantly the address of the BHS contract instance where this binary is registered and its Hash ID. It is important to note that hashes stored in BHS contract instances are computed over binary and metadata, thus protecting the integrity of both.

Please recall from Subsection 2.2.1 that an Ethereum client can obtain and validate data pertaining to specific smart contract instances either by holding and maintaining a complete copy of the Ethereum blockchain and world state as a full node or by requesting data from full nodes when running as a light client. Similarly, recall from Subsection 2.2.2 that Ethereum clients can be instructed to search old blocks or monitor new blocks for specific events emitted during transactions. Both of these functionalities are crucial for a Palinodia client to fulfill its purpose.

Installation: When a user installs a binary of a given software for the first time, their Palinodia client extracts the aforementioned metadata and requests the current state of the referenced BHS contract instance from a locally running Ethereum client. After ensuring that the Hash ID of the downloaded binary is not revoked, the Palinodia client compares the hash retrieved by the Ethereum client to a self-computed hash over binary and metadata to verify their integrity and endorsement. Next, the Palinodia client requests the current state of the SW contract instance referenced in the previously obtained BHS contract instance's state to ensure that the latter is properly registered in the former. In the event that any of these checks fail, the Palinodia client alerts the user not to continue with the software installation. Lastly, as this is the first time this Palinodia client encounters this root software identity, it asks the user whether or not to add it to its list of trusted software identities. This trust-on-first-use mechanism involving the user is necessary as there is currently no reliable way for a Palinodia client to distinguish a software's correct identity representation from maliciously created counterfeit identities. Note that the address of the SW contract instance is stored as a trust anchor. In this way, an established trust relation between user and software persists even if ownership and control over a given software identity changes. Such an ownership change is documented on-chain, as described above, and is communicated to users through their Palinodia client.

After a binary is installed as described above, the Palinodia client instructs its local Ethereum client to monitor both the SW and BHS contract instances for important events that signify new releases, revocations of prior releases, or the deregistration of the BHS contract instance, for example.

Update: When a user updates a previously installed software binary, either prompted through an update event or another mechanism, integrity and revocation checks are performed the same way as during initial installation. However, after the Palinodia client traces the updated binary back to its root software identity, it either recognizes the SW contract instance's address as previously trusted or it initiates another trust-on-first-use loop with the user if the instance is unknown. In the latter case, an attentive user should become suspicious, especially if the software name stored in the unknown SW contract instance matches the name of one of its trusted software identities.

Revocation: During a `revokeHash` transaction, a `Revocation` event containing the revoked Hash ID as an indexed parameter is emitted. Ethereum clients running at the time such a transaction is added to the Ethereum blockchain recognize this event while processing new blocks based on the included Hash ID that was set up for monitoring by the Palinodia client during installation or update of the corresponding binary. The Ethereum client then forwards the revoked Hash ID to the Palinodia client to communicate the revocation of a currently installed binary to the user. Palinodia and Ethereum clients that were offline at the time a revocation was published will examine blocks added to the Ethereum blockchain since they were last online and process `Revocation` events as described above.

3.5. Evaluation

To gauge the practicality of Palinodia, we implemented all smart contract types described in Subsection 3.4.1 in Solidity 0.8[2] and deployed them to a local development blockchain running the London hard fork using Ganache v7.2.0 of the Truffle Suite development tools. We then measured gas costs for contract deployment as well as common operations. Similarly, we implemented a Palinodia client in Go and measured its resource consumption and performance. To measure the time between the issuance of a revocation and a monitoring Palinodia client alerting its user, we conducted tests on the Ropsten testnet.

We first provide a quantitative evaluation of Palinodia's on- and off-chain components before semi-formally discussing security considerations, particularly regarding attacks described in Subsection 3.3.2.

[2] `https://git.scc.kit.edu/dsn-projects/dissertations/dsim/-/tree/main/Palinodia/`

Table 3.4.: Gas costs of Palinodia contract deployment and common operations. Conversion of gas to USD via daily average exchange rates for 1st June 2022 as reported by Etherscan: USD 1817.42 per ETH, ETH 60.06×10^{-9} per gas.

Operation		Gas	USD
SW contract			
Deployment		1 143 544	124.81
`setSoftwareName`	5 char Name	41 965	4.58
	10 char Name	42 025	4.59
`registerBHSContract`		135 838	14.83
`deregisterBHSContract`		50 573	5.52
BHS contract			
Deployment		1 077 213	117.57
`setSDP_ID`	5 char SDP ID	79 745	8.70
	10 char SDP ID	79 805	8.71
`publishHash`	5 char Hash ID	89 607	9.78
	10 char Hash ID	89 667	9.79
`revokeHash`	5 char Hash ID	38 789	4.23
	10 char Hash ID	38 849	4.24
IDM contract			
Deployment		927 201	101.20
`addIdentity`		99 050	10.81
`removeIdentity`		34 996	3.82
`changeCertificateCID`	46 char CID	42 287	4.62
	60 char CID	42 455	4.63
	63 char CID	47 731	5.21
	111 char CID	53 475	5.84

3.5.1. Gas Costs & Performance

Please recall from Subsection 2.2.3 that deploying and executing smart contracts on Ethereum incurs costs in the form of gas, which in turn is converted from and to Ether based on an exchange rate set by a transaction's sender. Table 3.4 provides a comprehensive overview of the deployment and execution costs of Palinodia contracts. An important aspect to consider when interpreting the costs of Table 3.4 is the frequency with which certain operations are performed. Comparatively costly deployments of contract instances are only performed rarely. More specifically, only one SW contract is deployed per software identity whereas the number of BHS contract instances depends on the organizational needs and distribution paths of a given software. Each actor or mutually trusted group of actors needs their own IDM contract instance. As mentioned previously, IDM contracts act as a stand-in for a more comprehensive user identity management system that a user would have already deployed before using Palinodia. The more frequent operations of hash

Table 3.5.: Performance of the Palinodia client and Geth (v1.9.22, default settings) in light and full synchronization mode on the same device connected to the Ropsten testnet (as of Oct 2020) [82]. **Rows**: *Incoming* network traffic for validating a binary (NT-V) and maintaining synchronization with the blockchain (NT-M); time spent by respective client during validation (TTV) and revocation (TTR) of a binary. Timing of revocation begins with issuance and includes time for the transaction to be included in a block and propagated to Geth. NT-V, TTV, TTR averaged over 100 binaries.

Client	Palinodia	Geth (light)	Geth (full)
RAM	40 MB	275 MB	1.4 GB
Disk	70 MB	500 MB	90 GB
NT-V	0 B	1.5 MB	0 B
NT-M	0 B	12 MB d^{-1}	450 MB d^{-1}
TTV	4 ms	400 ms	13 ms
TTR	4 ms	20 s	20 s

publication and revocation incur reasonable costs compared to the guarantees provided by the Ethereum ecosystem as we describe in the next section.

It is important to note that costs for operations like `setSoftwareName`, `publishHash`, and `changeCertificateCID`, that store a string of variable and user-chosen length, depend not only on the amount of new data being stored but also on the data previously stored in the corresponding contract variable. Writing to a variable for the first time is more expensive, whereas freeing up space by replacing a string with a shorter one reduces costs. In Table 3.4, we report costs for replacing strings of equal length, where applicable. The lengths of CIDs for `changeCertificateCID` are chosen based on possible IPFS CID lengths, as we describe in Section 2.3.

When evaluating the performance of the Palinodia client, it is important to keep the Ethereum client's role as gateway to blockchain data in mind. In order to observe both clients' interactions in a practical environment, we set up Go Ethereum (Geth) in either full or light synchronization on the Ropsten test network together with a Palinodia client on a single machine. At the time of our measurements, the Ropsten test network was a Proof-of-Work-based blockchain identical to the Ethereum main chain in terms of protocol, EVM, and 13 s block time with the crucial difference that Ether on Ropsten can be obtained easily through so called faucets and it holds no monetary value. Ropsten was successfully transitioned to Proof-of-Stake in June 2022 in preparation for The Merge of Ethereum's main net. Running tests like ours on Ropsten rather than main net is both ethically and financially advisable.

Through our setup, we were able to monitor memory, disk, and network resource utilization during binary validation, continuous monitoring, and revocations. For revocations, we particularly wanted to measure the delay between issuance of a revocation by a software developer and a Palinodia client of a corresponding software user processing it. After setting up Palinodia contract instances on Ropsten and adding them to the Palinodia client for monitoring through Geth, we issued revocation transactions to Ropsten from the same machine using the Infura API[3], thus allowing us to both time the delay accurately and ensure that Geth could only learn of revocations via Ropsten. We provide the results of our measurements in Table 3.5. Please recall from Subsection 2.2.1 that, broadly speaking, Ethereum clients can operate in a light and full synchronization mode, the trade-offs of which are both evident and relevant in the above table. In light mode, Geth only receives and validates block headers, which suffices to perform event monitoring related to new publications or revocations of binaries in Palinodia instances with a comparatively low disk footprint. However, to obtain state information of a particular contract instance, a Geth light client must request it from full clients it is connected to, resulting in approximately 1.5 MB of incoming network traffic and a 400 ms delay during the validation of a new binary. A Geth client with full synchronization already holds all state information, resulting in a much larger disk footprint, but allowing it to respond rather quickly during binary verification without any additional network traffic. Note that we performed the above measurements on the Ropsten test net, which gives a useful comparison between resource demands of light and full sync modes but does not accurately reflect the demands a user would face when connecting to the Ethereum main net. On the Ethereum main net, the disk space requirements for a Geth light node are below 500 MB[4] whereas a full synchronization demands roughly 750 GB as reported by Etherscan[5] at the time of writing. Regarding the propagation of revocations, it is important to note that the time for a corresponding transaction to be included in a new block is included in the 20 s delay for both light and full Geth clients. Ropsten aims for the same 13 s block time as the Ethereum main net, but the competition of new transactions for limited block space is significantly lower than on main net, making for much more predictable overall revocation delays. Consequently, the 20 s we measured above should be taken as a best-case lower bound.

3.5.2. Security Considerations

We now describe how Palinodia derives its security properties from its construction and the underlying Ethereum blockchain and peer-to-peer network, beginning with the attacks described in Subsection 3.3.2. For this evaluation, we assume users only install and use binaries registered in Palinodia smart contract instances.

Arbitrary binary: An attacker that manipulates authentic binaries to compromise user systems fails at the integrity checks performed by the Palinodia client before a binary is installed or executed. To overcome this hurdle, an attacker may deploy their own Palinodia contract instances and register their manipulated binaries just like a legitimate software developer or maintainer would. The success of this approach falls back to the trust-on-first-use loop performed when a Palinodia client encounters a root software identity for the first time. If a user expects a prompt asking whether to add a new software identity to the set of trusted identities, an attacker can succeed with a counterfeit Palinodia instance. Even

[3] `https://docs.infura.io/infura/networks/ethereum`
[4] `https://ethereum.org/en/developers/tutorials/run-light-node-geth/`
[5] `https://etherscan.io/chartsync/chaindefault`

worse, with such a counterfeit software identity in the set of trusted identities of a user's Palinodia client, an attacker could deliver updated versions of their manipulated binaries without raising suspicion. However, if a user is not expecting a trust-on-first-use prompt because they are supposedly updating a software they already installed, this attack would fail. Lastly, an attacker could attempt to add maliciously altered binaries to the legitimate software identity by compromising the necessary key pairs of software developers or maintainers. Since the attacker leaves plainly obvious proofs of their actions in the form of transactions on a public ledger, which also emit easily monitorable events enforced through smart contract code, such an attack only has a brief window of opportunity to succeed. Through the root owner mechanism in all Palinodia contracts, the compromised software developers and maintainers are able replace the affected key pairs, repair the damage caused by an attacker through revocations and updated releases, and reestablish control over their software identities.

Replay attack: An attacker that attempts to compromise user systems by presenting authentic, unmodified, but outdated and vulnerable binaries as current fails at the revocation checks performed by the Palinodia client, as long as the responsible software maintainer revoked the corresponding Hash IDs in their BHS contract instance. Note that this is the case regardless of whether or not the corresponding software identity is part of the set of trusted identities within a user's Palinodia client. Altering the view of a user's Ethereum client on the blockchain state in order to hide a particular revocation from both the Palinodia client and the user challenges the fundamental security assumptions of a public blockchain system like Ethereum. First, an attacker would have to eclipse a victim by controlling all connections they have to the Ethereum peer-to-peer network. Then, assuming that the victim was eclipsed ahead of a particular revocation, an attacker would have to generate an alternate valid block and subsequent blocks in order to successfully convince the user's Ethereum client that the revocation never happened. This would currently require mining power on par with all honest Ethereum miners combined in order to keep up with the block creation times the Ethereum client expects. Undoing a revocation an Ethereum client has already received and processed requires even more mining power as an attacker would have to generate an alternate blockchain that is longer than the canonical chain the client already knows. Both of these attacks violate our assumption on the mining power available to attacks as described in Subsection 3.3.2. Lastly, compromising key pairs is not helpful for this attack as there is no functionality in BHS contracts to undo a revocation, even for root owners.

Freeze attack: Similar to replay attacks described in the previous paragraph, an attacker trying to prevent a user's Ethereum client from obtaining information on the current state of software identities or attached binaries would encounter similar problems. The Ethereum protocol currently aims at generating a new block every 13 s. An Ethereum client that does not receive a new valid block for several minutes would raise suspicion that something was wrong and the user should investigate. In this context, it is also worth pointing out how new blocks certify the freshness of software identity information even if no changes happened. Please recall from Subsection 2.2.2 that part of every block header is the state root which is computed over the entire Ethereum world state and is used by Ethereum clients when obtaining and verifying contract instance states. In this way, a contract instance's state can be certified and verified as current by cryptographically proving its inclusion in the state root contained in the most recent block header, regardless of when it was last modified through a transaction. Another approach an attacker could take to execute a freeze attack is to prevent software developers or maintainers from issuing transactions via an eclipse attack. However, similar to scenarios described above, such an attacker would have to either fabricate a counterfeit blockchain to make it seem like transactions were executed, or raise suspicion when transactions repeatedly fail to be recorded on the canonical Ethereum blockchain. Publicly available block explorers like Etherscan[6] can also be used by software developers, maintainers, and users alike to access current information on the blockchain as a whole or individual contract instances and transactions.

3.6. Discussion

Based on our results, we can answer the first of our research questions stated in Chapter 1 affirmatively. While further uses for software identities on distributed ledgers are explored in Chapter 5, Palinodia serves as a necessary foundation and shows how the requirements for software identities described in Section 2.1 and the properties of Ethereum are well-aligned: The Ethereum blockchain serves as a tamper-evident, strictly access-controlled, and therefore reliable source for information on software identities, while the peer-to-peer network ensures a high availability of this data source and fast dissemination of updates, like new releases or revocations.

The properties of revocations in Palinodia are worth emphasizing in comparison to existing approaches like signed binaries and centrally-hosted hash-based verification. Signatures attached to binaries, while not necessarily establishing a software identity that connects together multiple binaries, do provide integrity protection and endorsement by an authoritative source with perfect availability as they are distributed alongside binaries. However, systems relying on signatures attached to binaries exhibit unreliable revocations [48] resulting, in part, from one developer certificate being used to sign multiple binaries and a decoupling between the validity of said developer certificate and signatures for binaries created with it. Centrally-hosted hash-based verification servers, meanwhile, allow for very precise revocations of individual binaries but may not provide a reliably high availability as signed binaries. Additionally, such verification servers must be protected against compromise, otherwise an attacker can legitimize manipulated binaries by adding their hashes to the respective servers. Using the Ethereum blockchain, Palinodia achieves both precise and highly available revocations. While deployment and transaction costs for Palinodia smart contracts are significant, they effectively replace the effort of establishing and maintaining a verification server with "renting" this functionality from the Ethereum ecosystem. At the time of writing, reading data from the Ethereum blockchain does not incur any costs, which is rather beneficial for an application like Palinodia with a very

[6] `https://etherscan.io/`

skewed read and write demand: Each binary hash is written and possibly revoked only once but read many more times, depending on how many users a software has.

In addition to the security properties described in Subsection 3.5.2, there are also more incidental benefits to Palinodia's construction on a public ledger that are worth discussing. For example, in centrally-managed open-source software repositories like the Node Package Manager (npm)[7], control over well-established packages can be transferred by the repository operator when they are abandoned by their previous owners. Ohm *et al.* [67] report that such ownership transfers have already been used by malicious actors to take over and weaponize established open source packages in order to attack and compromise user systems. While Palinodia can not prevent the abandonment of software identities, it does prevent abandoned identities from being claimed by or reassigned to anyone without the previous owner's explicit consent. Note that open source projects using Palinodia could still be forked to establish their own software identity, a process that would be plainly visible to both users and developers alike. Specifically, trust relations between users and an abandoned software identity would not immediately transfer to the software identity of a forked and revived software but require explicit action by users. This example succinctly demonstrates one qualitative difference between software identities that are managed by a centralized authority and self-sovereign software identities like those provided by Palinodia.

While implementing Palinodia, we made two design decisions that are worth highlighting here. First, we originally suggested that a Palinodia client could obtain and use pending revocation transactions to further reduce the delay between their issuance and user notification. This turned out to be both technically challenging and of questionable value. On the technical side, Ethereum light clients are not meant to request and process pending transactions from full nodes, so the protocols used in this interaction do not support this functionality. Full nodes, however, can obtain pending transactions and be instructed to forward them to a Palinodia client for further processing, but their operation is rather demanding as we describe in Subsection 2.2.1. Additionally, the usefulness of pending revocation transactions is somewhat diminished by the fact that they may not end up being persisted on the Ethereum chain and thus not be considered actually executed. For example, if the issuing address of a revocation is deauthorized in the corresponding IDM contract instance before the revocation is processed or the transaction is invalidated by its sender because it was issued by mistake, it would not be persisted on-chain. These scenarios reinforce the recommendations we noted in Subsection 2.2.3 that applications relying on transactions recorded on-chain should wait for seven or more subsequent blocks so that a chain reorg is less likely to undo them.

The second design decision concerns the structure of on-chain components and inherent trade-offs. For example, we opted to deploy one BHS contract instance per software distribution platform. Alternatively, one single BHS contract instance could be constructed that is shared by multiple software maintainers, none of whom would have administrative privileges over the entire instance. This would result in a more complex contract code that

[7] `https://www.npmjs.com/`

would only have to be deployed once per software identity, but it would also increase the costs for publishing hashes as additional information would have to be stored to support the more fine-grained access control necessary for such a multi-tenant smart contract.

3.6.1. Limitations & Future Work

Our primary focus with Palinodia was to demonstrate feasibility of software identity management on a distributed ledger at the example of Ethereum. As such, aspects like security and contract instance reusability were not observed to the extent of a production-level deployment. For example, deployed BHS contract instances can be "maliciously captured" by SW contract instances other than the one intended by a software maintainer. Similarly, deregistered BHS contract instances are currently left in an unusable state as the deregistration is only processed within the SW contract instance. A general design pattern for interlinking smart contract instances under the control of different actors that protects against unintended links and provides a well-defined linking and unlinking procedure could be interesting future work.

Similar to the point above, our measurements regarding the performance of the presented Palinodia client does not consider how the Ethereum peer-to-peer network would cope with a large-scale use of Palinodia, particularly the number of light clients making infrequent read requests to full nodes. While we made anecdotal observations during our experiments that light clients had difficulties finding and connecting to full nodes configured to serve them, a quantitative measurement of the "light client acceptance" of Ethereum's peer-to-peer network could be valuable future work to judge the practicality of applications such as Palinodia. Such measurements would be particularly interesting after Ethereum's switch to Proof-of-Stake as part of The Merge and the planned introduction of data sharding meant to better support applications like ours.

There is also a privacy aspect worth highlighting with the use of Ethereum light clients in Palinodia. While the delegation of monitoring requests from light to full nodes is somewhat privacy-preserving through the use of probabilistic Bloom filters, requests for specific transactions or smart contract instance states eventually leak to the full node which software the user behind a particular light client is using. Once enough software identities are in use, full nodes may be able to fingerprint and reidentify light clients based on their monitoring and retrieval requests. Assuming that light-client serving full nodes become more abundant as time goes on, developing connection management strategies to preserve a user's privacy as much as possible could be interesting future work, both on Ethereum in particular or on public DLTs in general.

Lastly, each Palinodia software identity currently represents a software in isolation, even though software of sufficient complexity is rarely self-contained and may instead include libraries or modules. A logical next step would therefore be an extension to Palinodia such that relations and dependencies between different software identities can be expressed and recorded as well. Such a feature would be particularly relevant with regard to security-related revocations as software that depends on a revoked version of a library, for example, could be identified more easily.

4. ETHTID: Threshold Information Disclosure on Ethereum

The content presented in this chapter has been published previously in the paper titled "ETHTID: Deployable threshold information disclosure on Ethereum" by Stengele, Raiber, Müller-Quade, and Hartenstein presented at the 3rd international conference on blockchain computing and applications [87]. An extended preprint with the same title and by the same authors is available on arXiv [86].

Public distributed ledgers like Ethereum provide a remarkable set of features as we explained in Chapter 2 that can be leveraged for applications such as Palinodia, as we showed in the previous chapter. However, coordinating the disclosure of a set of statements is a functionality that, at least to our knowledge, no ledger had an inherent need for and it is therefore not immediately available on any one of them. In recent years, the growing extent of "front running", i.e. the use of non-public information in the form of pending transactions for one's own financial benefit, particularly by miners in distributed ledgers, has sparked work to amend this functionality to existing systems [92]. While the problem we tackle in this chapter differs in its setting and scope to the front running scenario above, the core problem of managing the release of information in a decentralized way is similar. Please recall from Chapter 1 that one of our overall goals is to augment software identities, as described in the previous chapter, with independently verified attributes regarding specific characteristics, which is the focus of the next chapter. In order for such a review process to be independent, a coordinated disclosure of results is essential. Similarly, to prevent reviewers from committing to multiple results and only selectively disclosing some or none of them depending on the results of others, control over said disclosure should be delegated. A trusted third party can provide such a functionality rather easily, but since we aim to build a fully decentralized system, the focus of this chapter is to establish a decentralized disclosure coordination mechanism.

4.1. Problem Statement

We propose the coordinated disclosure problem independent of our use case as follows:

> Each party $p_i \in \{p_1, p_2, \dots, p_n\}$, with n unknown in advance, commits to message m_i at time $\mathrm{t}_i^{(\mathrm{c})}$. The public disclosure of the contents of messages m_i at time $\mathrm{t}^{(\mathrm{d})} > \mathrm{t}_i^{(\mathrm{c})}, i \in \{1, 2, \dots, n\}$ is *coordinated* if the following properties hold:

- **Fairness**: If the contents of any message m_i are disclosed, then the contents of all messages m_j, $j \neq i$ are disclosed.
- **Hiding**: Between commitment at time $\mathrm{t}_i^{(c)}$ and disclosure at time $\mathrm{t}^{(d)}$, the contents of message m_i are only known to party p_i.
- **Binding**: After commitment at time $\mathrm{t}_i^{(c)}$, the contents of message m_i are immutable.

The one-to-one mapping of messages to parties was chosen for readability and is not strictly required or enforced. Indeed, each party p_i can commit any number of messages to be disclosed at time $\mathrm{t}^{(d)}$. A corollary property that immediately follows from the fairness and binding property is that, once committed, the disclosure of individual message contents cannot be prevented, not even by their respective authors.

One important aspect we need to address at this point is the ability of each p_i to withdraw from a coordinated disclosure after committing messages and instead reveal their contents prematurely and unilaterally. We see this as a fundamental and unavoidable limit as each party knows what they committed to and they cannot be prevented from disclosing this information. Crucially, such a deviation is mainly to the detriment of the deviating party and does not affect the coordinated disclosure of other parties' message contents. For the sake of readability, we will omit the explicit mention of this caveat going forward.

In addition to the functionality described above, an equally important goal of our approach is to record and time-stamp the disclosure process such that the independent generation of all m_i can be judged afterwards based on their time stamps. Since we intend to use our disclosure mechanism on a distributed ledger, this time-stamped recording can be achieved rather gracefully.

We introduce the concept of threshold information disclosure (TID) to realize a coordinated disclosure mechanism on Ethereum. This approach revolves around a temporally decoupled asymmetric key pair: An encryption key is provided early to enable the encryption and commitment of messages whereas the decryption key is released at a specified later time to realize the disclosure of message contents. Fairness is achieved by making access to all committed message contents dependent on a singular decryption key. To fulfill the hiding property, this decryption key cannot be generated or held by a single party. We therefore task a council with performing a distributed key generation procedure so that the decryption key is held in a shared state among all council members with no single member having access to the key itself. The council is also tasked with recovering said decryption key for its scheduled release. Lastly, to avoid single points of failure, i.e. the disclosure being dependent on any single council member, we employ threshold secret sharing of the decryption key, such that only a configurable portion of the council, the eponymous threshold, must collaborate in order to recover and publish it. Essentially, threshold information disclosure delegates and decentralizes the privilege and responsibility to grant public access to an arbitrarily large set of appropriately prepared messages.

The remaining chapter is structured as follows. In Section 4.2, we discuss related work, both in terms of cryptographic approaches and alternative realizations. With Section 4.3, we provide distinct fundamentals for this cryptography-focused chapter which includes introducing the threshold secret sharing scheme by Shamir and Feldman's use of it in a verifiable manner as crucial parts of threshold information disclosure. We provide our system model, including roles as well as attacker and trust assumptions in Section 4.4. In Section 4.5, we present ETHTID, our implementation of threshold information disclosure on Ethereum, before we evaluate it in Section 4.6. We discuss our findings before examining limitations and highlighting possible future work in Section 4.7.

4.2. Related Work

We cover related work in two distinct directions: First, we briefly explain available cryptographic primitives and the rationale for our selection. Second, we review and discuss other approaches to achieve a coordinated disclosure functionality and how our work relates to them.

Based on our approach of using an asymmetric key pair to facilitate coordinated disclosure, an early choice in design concerns the underlying cryptosystem. While distributed key generation (DKG) for factoring-based cryptosystems like RSA exist [13], their probabilistic multi-round structure is significantly less efficient than DKGs for discrete-log-based cryptosystems that enable Diffie-Hellman(-Merkle) key exchanges [26, 59] or ElGamal encryption [28], which have a constant number of rounds. In particular, we employ a variation of the verifiable secret sharing (VSS) scheme by Feldman [30], which consists of a constant number of communication rounds. In his paper, Feldman cites a prominent threshold secret sharing scheme by Shamir [80], which we also adopt and introduce in more detail in Section 4.3. However, it is instructive to briefly examine other threshold secret sharing schemes and explain why we opted not to use them.

Based on the Chinese remainder theorem, Asmuth and Bloom [3] and Mignotte [60] independently proposed threshold secret sharing schemes where an integer is decomposed into a system of simultaneous congruences, with each congruence being a share of the secret. Through careful selection of moduli and other parameters, it is ensured that a threshold number of congruences are required to recover a shared secret and fewer congruences reveal no useful information about it. Kaya and Selçuk present a joint random secret sharing scheme based on Asmuth Bloom [47], which is unsuitable for our application for two reasons: First, it requires a shared RSA modulus, which we already discounted above. Second, the size of shares grows with the number of share holders and the configured threshold due to the necessarily large moduli, which is not ideal in an execution environment such as ours where the transmission and storage of every byte incurs costs.

Blakley presents a threshold sharing scheme based on affine hyperplanes in a vector space whose dimensionality corresponds to the desired threshold [11]. A shared secret is encoded as the intersection of these hyperplanes and due to the dimension of the overall vector space, one needs the designated threshold number of them for recovery. Distributed key generation schemes based on the threshold secret sharing scheme by Blakley have only recently started to appear [73]. However, similar to DKG schemes for Asmuth-Bloom from above, the size of shares grows with the threshold as it determines the dimensionality of the underlying vector space, making them similarly unfitting for our application.

We now review alternative and related approaches to coordinated disclosure. When comparing previous works to our approach, it is instructive to consider the number of mutually distrusting senders, i.e. they do not wish to disclose their messages to each other prematurely, and the number of possible recipients of eventually disclosed messages. ETHTID enables a many-to-many disclosure without a trusted third party.

An early exploration of encrypting messages “to the future” was submitted by May to the Cypherpunks mailing list in 1993[1]. Rivest, Shamir, and Wagner then introduced this notion to the scientific literature in 1996 [77]. Rivest *et al.* proposed time-lock puzzles that can be efficiently generated by a sender but require a configurable amount of computational work to be solved by a recipient, which is estimated by the sender to take a certain amount of wall clock time. These puzzles can be attached to encrypted messages such that the puzzle’s solution grants access to the message. The concept was later improved by Mao [54] and further studied by Mahmoody *et al.* [52], among others. In their initial form, time-lock puzzles based on computational effort constitute a one-to-one disclosure mechanism: Having multiple senders use the same time-lock puzzle for a coordinated disclosure either requires a trusted party to generate the puzzle and encrypt messages on behalf of their senders, or all senders hold the same key, meaning they gain premature access to each other’s messages. Recipients face a similar problem to access a set of time-lock encrypted messages. All interested recipients have to independently exert computational effort to solve the time-lock puzzle unless or until one of them shares the solution with everyone else. Without a benevolent recipient to share the solution, it is highly likely that recipients solve a given time-lock puzzle at different times, thus leading to a rather loosely coordinated disclosure. Malavolta and Thyagarajan [53] present an approach for homomorphically combinable time-lock puzzles. With their approach, multiple inputs to an arbitrary function can be time-lock encrypted, the resulting puzzles can then be combined homomorphically into a single puzzle that, once solved, reveals the function’s output. Most crucially, the difficulty of the output puzzle does not depend on the number of inputs. While Malavolta and Thyagarajan describe applications for their scheme that require an algorithmic transformation of inputs, their approach may also be applicable to the coordinated disclosure problem.

Another general approach is to use trusted time servers, which release cryptographic information at specified times, thereby enabling the decryption of messages that were specifically prepared to be disclosed at those times. Rivest *et al.* [77], in addition to time-lock puzzles as described above, also sketched this idea in their paper. The concept was later improved by Di Crescenzo *et al.* [24] and Catholo *et al.* [20]. However, their focus revolves mostly on anonymity notions between senders, recipients, and the trusted time

[1] `https://mailing-list-archive.cryptoanarchy.wiki/archive/1993/02/a421c6fc805dfb4ae4197521e8a9e91dd456e3deab855f12af31a4b1ccccf6cb/`

server mediating the disclosure and less on coordinating the public disclosure of messages by multiple senders as we do. Nevertheless, many-to-one disclosures are straightforward with these schemes and by creating multiple encrypted messages for multiple recipients, many-to-many disclosure is also possible. However, the number of encrypted messages in this approach scales with the product of senders and recipients, posing a challenge to scalability in addition to relying on a centralized trusted third party.

Within the context of distributed ledgers, the work of Benhamouda *et al.* [10] is noteworthy in relation to our approach. They propose to distributedly generate and threshold-share a single secret and have the parties actively maintaining a distributed Proof-of-Stake ledger, in their case Algorand[2], indefinitely re-share this secret across changing committees as part of the underlying consensus protocol. In this way, a ledger itself could act as a cryptographic entity by generating signatures or decrypting information under specified conditions. Benhamouda *et al.* suggest that functionalities such as the one we establish here could be made available in their system as “threshold cryptography as a service”. However, as far as we could tell, their concept has not been implemented in Algorand and thus no derived features are available at the time of writing. The way in which such a service could be derived from an indefinitely re-shared secret is also left as future work.

Lastly, the work of Schindler *et al.* [79] is of particular relevance as we build upon it and adapt it to our use case. With EthDKG, Schindler *et al.* describe a framework for distributed key generation using Ethereum smart contracts for coordination, communication, and arbitration of disputes in case of misbehavior. They provide an implementation consisting of a Solidity smart contract and a Python off-chain application to execute a DKG for a Boneh–Lynn–Shacham (BLS) group signature scheme [15, 14]. Once established, a sufficiently large subset of a group can generate signatures that can be verified against the group’s public key, which is available on the Ethereum blockchain. In their use case, recovery of the shared secret key is neither necessary nor intended. We adopt the implementation of EthDKG, extend it with a coordinated recovery and publication of the shared secret key, and optimize it based on the requirements of our use case, allowing us to save significant costs.

4.3. Fundamentals

Since we focus more on cryptographic constructions in this chapter, we provide separate fundamentals here. We first review finite cyclic groups as the fundamental mathematical object and successively introduce notation and concepts necessary for the remaining chapter.

[2] `https://www.algorand.com/`

4.3.1. Notation & Number Theory

A group is defined as a set $\mathbb{G}$ and a binary operation $\cdot$ on its elements with the following properties:

- **Closure**: For all elements $g_1, g_2 \in \mathbb{G}$, $g_1 \cdot g_2 \in \mathbb{G}$.
- **Existence of an identity**: There exists an *identity*[3] $\mathfrak{e} \in \mathbb{G}$ such that for all elements $g \in \mathbb{G}$, $\mathfrak{e} \cdot g = g = g \cdot \mathfrak{e}$.
- **Existence of inverses**: For all elements $g \in \mathbb{G}$, there exists an element $g^{-1} \in \mathbb{G}$ such that $g \cdot g^{-1} = \mathfrak{e} = g^{-1} \cdot g$.
- **Associativity**: For all elements $g_1, g_2, g_3 \in \mathbb{G}$, $(g_1 \cdot g_2) \cdot g_3 = g_1 \cdot (g_2 \cdot g_3)$.

If the set $\mathbb{G}$ is finite, the group is called *finite* and $|\mathbb{G}|$ is the *order* of the group. For notational convenience, we denote applying the group operation m times to a group element g as exponentiation:

$$g^m = \underbrace{g \cdot g \cdot \dots \cdot g}_{m \text{ times}}.$$

It is straightforward to see that the conventional rules for exponentiation also apply, i.e. $g^m \cdot g^{m'} = g^{m+m'}$ and $(g^m)^{m'} = g^{mm'}$.

A finite group is *cyclic* if all of its elements can be described as powers of a particular element called the *generator*, denoted as g. Let $p = |\mathbb{G}|$ be the order of the group, then:

$$\mathbb{G} = \left\{g^0, g^1, g^2, \dots, g^{p-1}\right\}.$$

Naturally, $g^0 = \mathfrak{e}$. It is important to note that g and its powers are group elements whereas exponents are integers modulo p, or $\mathbb{Z}_p$ for short. Similarly, addition and multiplication in the exponent are also integer operations. Analogous to the use of exponentiation, the discrete logarithm for finite cyclic groups is also well defined and of central importance to corresponding cryptosystems. For a group element $h = g^x$, its discrete logarithm is $\mathrm{dlog}_g\, h = x$, i.e. the number of times g was applied to itself using the group operation to result in h. The security of discrete-log cryptosystems is based on selecting finite cyclic groups for which computing discrete logarithms is believed to be hard. Using the example above, given x, computing $h = g^x$ should be efficient whereas the inverse, deriving x from h, should be computationally infeasible.

4.3.2. Distributed Key Generation & Threshold Sharing

With notation and fundamental concepts covered above, we now turn to distributed key generation (DKG) and threshold sharing in discrete-log-based cryptosystems. On the surface, DKG in discrete-log-based cryptosystems is rather straightforward: Let $\mathbb{G}$ be a

[3] We use the Gothic font $\mathfrak{e}$ here to avoid a collision in notation later in the chapter.

cyclic group of order p with generator g in which computing the discrete logarithm is hard. In a council of size n, every member c_i draws a random $d_i \in \mathbb{Z}_p$, publishes $e_i := g^{d_i}$ and keeps d_i to themselves. The council encryption key is then $e := \prod_{i=1}^{n} e_i = g^{\sum_{i=1}^{n} d_i} = g^d$. However, in such a construction, every piece d_i would be required to recover d, thereby enabling any council member to prevent recovery and thus disclosure by simply not participating. By integrating threshold secret sharing into the key generation, this dependency can be alleviated to a configurable extent.

We introduce the primary functionality of threshold secret sharing below by using the auxiliary role of a centralized dealer, which will be obviated later. A dealer wants to share a secret s among n parties such that $t + 1$ of them need to cooperate to recover s and t or fewer parties can not learn anything useful about the secret s. To do this, the dealer generates a *share* of the secret s for each party. By construction, $t + 1$ correct shares suffice to recover s and any fewer shares reveal no useful information about it. The parameter t is the eponymous threshold. As mentioned in Section 4.2, we employ the threshold secret sharing scheme by Shamir within the verifiable secret sharing scheme by Feldman, both of which we introduce here.

In his seminal paper [80], Shamir proposed a secret sharing scheme using modular arithmetic on polynomials. To share a secret integer $s \in \mathbb{Z}_p$ among n parties, a dealer first chooses a finite field of prime order p. Next, the dealer embeds the secret s as the constant term in a polynomial of degree t with all other coefficients a_i drawn uniformly at random from $\mathbb{Z}_p$:

$$f(x) = s + a_1 x + a_2 x^2 + \cdots + a_t x^t \pmod{p}.$$

By evaluating their polynomial at distinct points x_i, the dealer generates the value $y_i = f(x_i)$ for party c_i. The share for party c_i is then $r_i = \langle x_i, y_i \rangle$ and is transmitted confidentially by the dealer. If $t+1$ share holders $i_1, \ldots, i_{t+1}$ exchange their respective shares and perform a Lagrange interpolation [89], they can recover the polynomial $f(x)$ and with it the shared secret s:

$$s = \sum_{k=1}^{t+1} \left(\prod_{\substack{l=1, \\ l \neq k}}^{t+1} \frac{x_{i_l}}{x_{i_l} - x_{i_k}} y_{i_k} \right). \tag{4.1}$$

Note that knowledge of only t shares conveys no useful information about the secret s: For any guessed last share, the above interpolation gives a unique polynomial and any value of s remains possible. As such, the secret sharing scheme by Shamir is information-theoretically secure: No amount of computing power would enable an adversary to know with certainty whether or not they guessed a set of valid shares and recovered the correct secret.

On its own, Shamir's scheme is vulnerable to malicious dealers that distribute inconsistent shares, which would lead to different secrets based on which $t + 1$ shares were used for recovery. Feldman [30] presents a scheme to use Shamir's secret sharing in a *verifiable* way by forcing a dealer to publicly commit to their polynomial in such a way that recipients of shares can check their correctness and overall consistency. To do this, a finite cyclic group of prime order p with generator g for which the discrete logarithm is hard to compute is

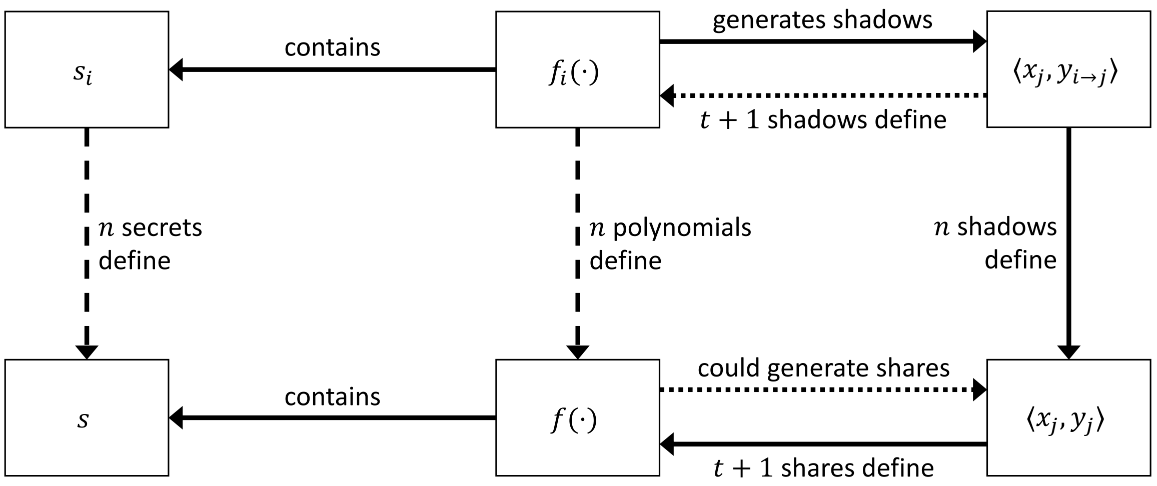


Figure 4.1.: Overview of cryptographic objects during threshold information disclosure, consisting of Feldman DKG, threshold sharing, and recovery. Solid arrows show necessary relations and operations. Dashed arrows signify implicit relations. Dotted arrows show operations that are possible but not necessary.

used. The dealer broadcasts commitments[4] to their polynomial coefficients $g^s, g^{a_1}, \ldots, g^{a_t}$ to all parties while transmitting secret shares r_i to each party confidentially. Each party c_i can then verify that the share $r_i = \langle x_i, y_i\rangle$ they received is correct by checking it against the previously broadcast polynomial commitments using the following equation:

$$g^{y_i} \stackrel{?}{=} g^s \prod_{k=1}^{t} (g^{a_k})^{x_i^k} \tag{4.2}$$

Notice that the polynomial of the share issuer is essentially reevaluated at the point x_i in the exponent on the right side of the above equation to result in the value y_i if the check passes.

By exploiting the fact that polynomials and their commitments[5] are additively homomorphic, Feldman's VSS scheme can be executed n times in parallel with each council member acting as dealer in one execution, which is commonly referred to as Joint-Feldman VSS. Essentially, each council member c_i threshold-shares their contribution s_i such that these n sharings can be combined into one sharing of the combined secret s. In this way, a council can generate s in threshold-shared form without a centralized trusted dealer and later recover and publish it to achieve threshold information disclosure (TID). We provide an overview of the cryptographic objects involved in TID and their relations in Figure 4.1 to accompany the following description.

[4] Note that we use the term "commitment" for the sake of readability. This construction does not constitute a commitment scheme in the strict cryptographic sense as it does not fulfill the hiding property: Given s, it is trivial to see that g^s commits to it. However, since all to-be-committed values in our construction are drawn randomly from a sufficiently large set and never explicitly revealed, this is not an issue.

[5] Note that polynomial commitments are elements of a finite cyclic group. Since we introduced the multiplicative notation above, the term "additive" is somewhat misleading here. However, combining two commitments with the corresponding group operation yields a commitment to their sum in the exponent: $g^a \cdot g^b = g^{a+b}$.

Each council member c_i threshold-shares a secret s_i among n council members, including themselves, as explained previously such that the resulting shares can be combined into a sharing of the sum of all secrets $s := \sum_{i=1}^{n} s_i$. For the sake of clarity, we use the term *shadow* $u_{i \to j}$ to refer to shares of the secret s_i of member c_i generated for member c_j. We reserve the term *share* r_i for shares of the group secret s that member c_i generates from shadows $\left\{u_{j \to i}\right\}_{j=1}^{n}$. Each member c_i embeds their secret s_i into a polynomial $f_i(x)$ as described above. In order for the shadows to be additively combinable, it is vitally important that all dealers generate the shadow for member c_j by evaluating their respective polynomials at the same value x_j, not doing so results in provably invalid shadows that will be handled in our implementation. Upon receiving shadows $u_{i \to j} = \left\langle x_j, y_{i \to j} = f_i(x_j) \right\rangle$ from all members c_i, including themselves, and verifying their correctness via Equation (4.2), member c_j can combine these n shadows into a share $r_j = \left\langle x_j, y_j = \sum_{i=1}^{n} y_{i \to j} \right\rangle$ of the group secret. By construction, $t + 1$ of the shares $\langle x_i, y_i \rangle, i = 1, 2, \cdots, n$ now uniquely define a polynomial $f(x)$ with $f(0) = \sum_{i=1}^{n} s_i = s$. Note that, before the combined secret s is recovered, none of the council members have access to it. TID concludes with the scheduled recovery and publication of the secret s by pooling $t + 1$ correct shares r_i and performing a Lagrange interpolation via Equation (4.1).

4.4. System Model & Assumptions

Before describing ETHTID, our implementation of threshold information disclosure on Ethereum, in detail, we provide a system, attacker, and trust model here and further specify our assumptions.

As we alluded to in Subsection 2.2.5, we treat Ethereum, more specifically its consensus and application layers as described in Subsections 2.2.3 and 2.2.4, as a synchronous, publicly accessible and viewable, authenticated, and reliable broadcast channel. In the synchronous communication model, sent (broadcast) messages are delivered with a known finite upper bound Δ. In our case, Δ describes the time between sending a transaction to the Ethereum peer-to-peer network and it being recorded on-chain and visible to anyone. Note that this is a model assumption that abstracts away some of the uncertainties of submitting transactions to be recorded on the Ethereum blockchain that depend on current transaction demand and fees, for example. We discuss the implications of this gap between model and reality further in Section 5.7. As we describe in Section 2.2, Ethereum as a whole is both accessible to and viewable by anyone. All transactions on Ethereum are cryptographically associated with their sending account, thereby ensuring sender authenticity and non-repudiation. Lastly, we model Ethereum as a reliable broadcast channel since it ensures through consensus that, once a transaction has been added to the blockchain, anyone can access it and, more importantly, everyone observes the same transaction.

4.4.1. Roles

Our system model contains the following roles as depicted in Figure 4.2:

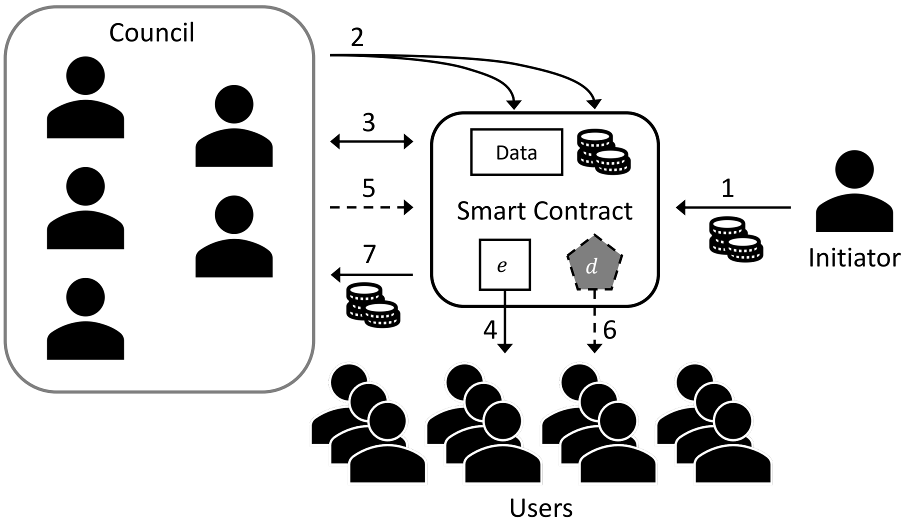


Figure 4.2.: System overview. **(1)** Initiator deploys smart contract with parameters and incentives. **(2)** Council members register with the contract by submitting data and a collateral. **(3)** Council members communicate through the smart contract to generate the encryption key *e* and establish a sharing of the decryption key *d*. In case of misbehavior, members submit a dispute to the contract to enforce consequences. **(4)** Users can obtain encryption key *e*. **(5)** At a codified time, council members reconstruct and submit decryption key *d* to the smart contract. **(6)** Users and the general public can obtain decryption key *d*. **(7)** Based on their behavior, council members receive a reward in addition to their collateral and are released from their obligation.

An *initiator* sets the council size, recovery threshold, reconstruction schedule, and deploys an ETHTID smart contract instance to orchestrate and coordinate a TID procedure. In practice, such an initiator would also provide an incentive for participation. As the party deploying a particular ETHTID contract instance, an initiator could use the opportunity to include malicious or deceitful functionality to the detriment of other parties. We discuss this aspect further in Section 4.7 and offer remedial measures in Subsection 4.7.1.

Council members facilitate threshold information disclosure by performing the distributed key generation, threshold sharing, and scheduled reconstruction of the encryption/decryption key pair. Council members would be required to submit a collateral that can be destroyed or redistributed in case of provable misbehavior. At the conclusion of a given ETHTID instance, each council member would reclaim their collateral in addition to their share of the reward provided by the initiator.

Users are the main beneficiaries of each ETHTID instance as the provided encryption key allows them to commit to the release of arbitrary messages. Other than obtaining the encryption key, once available, and using it to prepare their messages, users have no further interaction with an ETHTID instance. Indeed, once their encrypted messages are published in some way, users can become entirely inactive as the disclosure of their messages is then dependent on the publication of the corresponding decryption key.

Lastly, not depicted in Figure 4.2, the general public, including users who actively participated, can use a particular instance's decryption key after its publication to access the contents of previously committed messages. It is important to note that none of the above roles are mutually exclusive: An initiator can also be a council member and they can also commit to the disclosure of messages as a user. Problems only arise if too many council members are actually under the control of an attacker or they collude with each other, a notion we concretize in the next subsection.

4.4.2. Attacker and Trust Model

In our model, attackers can pursue two diametrically opposed goals in regards to disclosure: First, they can try to obtain the decryption key of a particular instance prematurely and either use the resulting exclusive access to committed messages for their own benefit or publish the key before it was scheduled to. Second, an attacker can attempt to prevent the coordinated disclosure entirely by ensuring that the decryption key cannot be recovered and published.

To accomplish these goals, an attacker can control or compromise up to the threshold t number of council members, thereby obtaining any secret information they hold and deciding their behavior throughout a given ETHTID execution, particularly their participation in the scheduled reconstruction of the threshold-shared decryption key. Similar to the previous chapter, attackers can also deploy and use smart contracts as well as participate in the consensus mechanism of the underlying blockchain. However, we must assume that attackers cannot gain control of the blockchain's consensus layer as that would allow them to prevent specific transactions from being executed, violating the communication model we outlined above. More generally, we assume attackers are not able to delay the execution of valid transactions long enough to force their senders to miss a contract-specified deadline. Similarly, we also assume that attackers cannot prevent read access to blockchain data indefinitely. Unlike transactions, read requests are not subject to time constraints and it is sufficient for our construction that they succeed eventually. Lastly, attackers are computationally bounded and cannot break cryptographic assumptions. In particular they are unable to compute discrete logarithms or find collisions in cryptographic hash functions.

Users trust that at least $t+1$ council members execute the coordinated disclosure procedure honestly: They participate correctly in the distributed key generation and sharing, issue disputes if they receive invalid shadows, provide their share of the decryption key to facilitate its scheduled recovery and publication, and they keep it secret until then. As in the previous chapter, initiators, council members, and users trust the parties maintaining the underlying blockchain in aggregate to follow and enforce its protocol, particularly the procedures and checks codified in smart contracts.

We note at this point the tension governing the choices of the threshold t in relation to the council size n with regards to preventing attacks and ensuring successful and timely recovery of decryption keys. With up to t council members compromised by an attacker and the assumption that at least $t + 1$ members follow the prescribed protocol honestly, we have an immediate lower bound of $n \geq 2t + 1$, which also results from a similar argument for the synchronous reliable broadcast communication model described above. However, as the compromise or control of t council members by an attacker represents a worst case scenario, larger and particularly smaller councils for a fixed threshold are still viable choices in practice with corresponding trade-offs: A larger council simplifies both intended recovery of the decryption key as well as compromise by an attacker, as more parties are available for either case. By contrast, a smaller council impedes both of these scenarios but also lowers the amount of members that can remain inactive before recovery and publication of the decryption key becomes impossible, thereby preventing a coordinated disclosure. At the extreme end of the latter option with $n = t + 1$, no redundancy remains as all shares are required for recovery and a single inactive, malicious, or compromised council member suffices to prevent recovery. This edge case is equivalent to the simple distributed key generation without threshold secret sharing from the beginning of Subsection 4.3.2.

4.5. ETHTID

We now present ETHTID, our concept and implementation of a threshold information disclosure mechanism as an Ethereum smart contract.

4.5.1. Overview

The interface of the ETHTID smart contract is described in Table 4.2. For convenience, we provide an overview of our notation in Table 4.1. As mentioned in the beginning of the chapter, our goal was not only to facilitate coordinated disclosure on a public ledger, but also to document the process such that disclosed messages can be judged as independently generated afterwards.

A crucial component of Ethereum that enables our implementation of ETHTID are *precompiled contracts* for addition and scalar multiplication of points on the Barreto-Naehrig (BN) elliptic curve [5]. Contrary to what the term implies, precompiled contracts are not actual smart contracts but rather optimized implementations of certain functions that were added to the EVM during hard forks to expand its capabilities. The term probably stems from the fact that precompiled contracts are called and used as if they were contracts deployed to specially reserved addresses. The precompiled contracts most relevant for our work were proposed in EIP-196[6] and deployed with the Byzantium hard fork in 2017. With the Istanbul hard fork in 2019, EIP-1108[7] was deployed, which reduced the gas costs for these precompiled contracts as their implementations had been further optimized in the meantime.

One aspect we omitted from our implementation to instead focus on demonstrating practical viability is incentivization: As each council performs a distributed key generation, sharing, and recovery not for their own benefit but as a service to users, a scheme to both reward participation and discourage misbehavior is needed in practice. Yakira *et*

[6] `https://eips.ethereum.org/EIPS/eip-196`
[7] `https://eips.ethereum.org/EIPS/eip-1108`

Table 4.1.: Overview of notation.

Symbol	Description
n	Size of council.
t	Threshold. $t + 1$ cooperating council members can reconstruct the shared decryption key.
$\mathrm{t}^{(\mathrm{d})}$	Point in time for disclosure via reconstruction and publication of the shared decryption key.
$\langle e, d \rangle$	Encryption and decryption key. The latter is threshold-shared among the council.
$\langle e_i, d_i \rangle$	Contributions of council member c_i to encryption and decryption key.
$\{A_{i,k}\}_{k=1}^{t}$	Commitments to polynomial coefficients of member c_i.
$u_{i \to j}, \overline{u_{i \to j}}$	Decrypted and encrypted shadow from council member c_i to member c_j.
r_i	Share of decryption key d held by council member c_i.
$\langle \mathrm{pk}_i, \mathrm{sk}_i \rangle$	Ephemeral key pair of council member c_i needed to generate symmetric key $k_{i,j}$. pk_i is submitted during registration.
$k_{i,j}$	Symmetric key between council members c_i and c_j, used to encrypt/decrypt shadows.
$\pi(k_{i,j})$	Zero-knowledge proof of correctness for $k_{i,j}$, submitted as part of dispute. Generation and verification depicted in Figures 4.8 and 4.9 respectively.

al. [91] provide a thorough game-theoretic framework that could, with some adjustments, be combined with ETHTID. Compared to ETHTID, the Escrow-DKG that Yakira *et al.* use to model their incentive scheme subdivides certain operations and has thus more opportunities to file complaints. Additionally, the authors also handle unjustified complaints, which are precluded in EthDKG and ETHTID through a cryptographic construction. In the following description of ETHTID, we note occasions where an incentive scheme would come into play and we discuss the consequences our cryptographic construction has on incentivization in Subsection 4.6.2.

4.5.2. Phase Structure

ETHTID proceeds in six phases as depicted in Figure 4.3, which we now describe in detail. We accompany the following description with pseudocode to outline the core operations of ETHTID from the viewpoint of a council member. A contiguous version of this pseudocode is provided in Appendix A. Figure 4.4 depicts the variables and arrays that define the state of each council member that will be used in all pseudocode excerpts in this section. Being executed via a public broadcast channel in the form of the Ethereum blockchain means that each pseudocode excerpt of ETHTID consists of two parts, one to

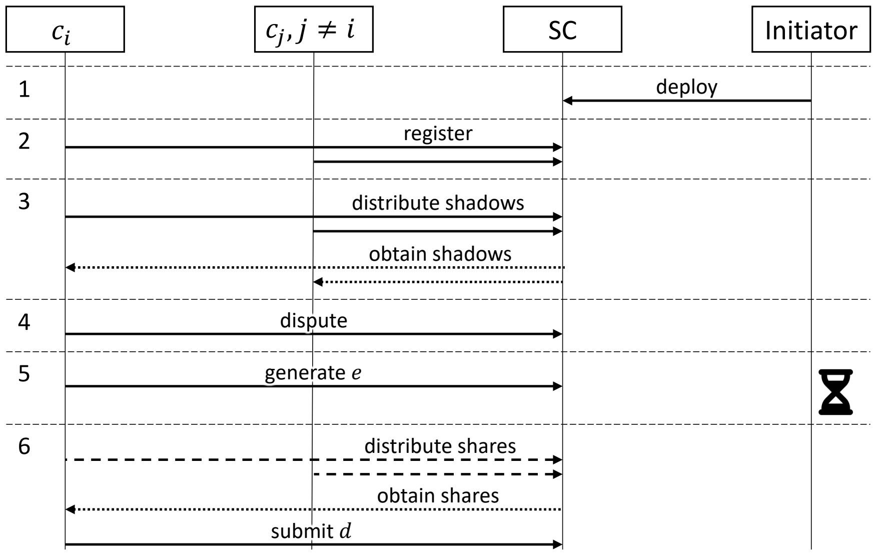


Figure 4.3.: Sequence diagram of one ETHTID execution with time going from top to bottom. One council member c_i is displayed separately from the remaining members c_j to signify transactions to the smart contract (SC) that only one council member needs to execute. Solid arrows show state-changing transactions, dotted arrows show read requests, dashed arrows show broadcast transactions. **(1: Initialization)** Initiator deploys smart contract instance. **(2: Registration)** Council members register with the contract instance and submit their ephemeral public key for the purpose of shadow distribution. **(3: Shadow Distribution)** Council members publish commitments to their respective polynomials and encrypted shadows for other council members. Likewise, council members obtain shadows encrypted for them and verify their correctness against polynomial commitments. **(4: Dispute)** In case a member receives an inconsistent shadow, they file a non-interactive dispute to disqualify the offending sender. **(5: Commitment)** Encryption key e can be generated based on member submissions from phase 3 that were not disqualified in phase 4. With encryption key e, messages can be committed for scheduled disclosure. This phase can last significantly longer than any other, depending on the schedule set by the initiator in phase 1. **(6: Disclosure)** Members submit their share of the decryption key d. Once enough valid shares are public, d can be recovered and submitted by one member. During its submission, the correctness of the decryption key d is verified against the encryption key e generated in phase 5.

```
t // Threshold codified in contract
members // Set of member indices
pks[] // Ephemeral public keys of other members for shadow encryption/decryption
sk_i // Own secret key for shadow encryption/decryption
enc_shadows[] // Encrypted shadows broadcast by other members
es[] // Contributions to encryption key of all members, including self
poly_comms[] // Commitments to polynomials of all members, including self
dec_shadows[] // Decrypted shadows necessary for generating own share
shares[] // Shares necessary for decryption key recovery
```

Figure 4.4.: List of variables and arrays defining the state of an ETHTID council member. Arrays are indexed by elements of the set members, which corresponds to the Ethereum addresses of council members.

Table 4.2.: Interface of ETHTID contract. Simple calls to retrieve values of attributes are omitted [86].

Name	Arguments	Functionality
Transactions		
`register`	pk_i	Council member commit to the participation in the protocol and submit ephemeral public key pk_i for the encryption of shadows.
`distribute_shadows`	$\{\overline{u_{i\to j}}\}_{j\neq i}, e_i, \{A_{i,k}\}_{k=1}^{t}$	Each council member c_i broadcasts encrypted shadows for other members and commits to their own polynomial for verification.
`submit_dispute`	$\{\overline{u_{i\to j}}\}_{j\neq i}, e_i, \{A_{i,k}\}_{k=1}^{t}, k_{i,j}, \pi(k_{i,j})$	Member c_j files dispute against member c_i. Contains `distribute_shadows` broadcast of member c_i. Correctness of key $k_{i,j}$ and invalidity of shadow $u_{i\to j}$ is verified before member c_i is disqualified.
`generate_e`	-	Derives encryption key e from submitted contributions e_i of qualified, i.e. not disqualified, members c_i.
`distribute_share`	r_i	Member c_i broadcasts their share of the decryption key d for reconstruction.
`submit_d`	d	Submit recovered decryption key d. Correctness is checked against encryption key e derived in `generate_e`.
Events		
`ShadowDistribution`	$\mathrm{sender}_{\mathrm{addr}}, \{\overline{u_{i\to j}}\}_{j\neq i}, e_i, \{A_{i,k}\}_{k=1}^{t}$	Broadcast of encrypted shadows and commitments during `distribute_shadows`.
`Dispute`	$\mathrm{sender}_{\mathrm{addr}}, \mathrm{accused}_{\mathrm{addr}}, k_{i,j}, \pi(k_{i,j})$	Valid call of `submit_dispute` by sender against accused.
`EK`	e	Publication of encryption key.
`ShareDistribution`	$\mathrm{sender}_{\mathrm{addr}}, r_i$	Broadcast of share by member c_i.
`DK`	d	Publication of decryption key.

prepare a broadcast and the other to receive broadcasts by other participants. We also introduce a slight change in notation here to improve the readability of these pseudocode excerpts. In the explanation of the Joint-Feldman VSS in Section 4.3, we emphasized the importance of a consistent mapping between participants and polynomial evaluation points for the purpose of generating shadows so that they can be combined into valid shares. We adopt from Schindler *et al.* [79] the ingenious approach of using the Ethereum addresses of registered participants for this mapping. These externally-owned addresses, which we briefly introduced in Subsection 2.2.3, are pre-established and easily accessible information for both smart contracts and participants alike and due to being generated via a cryptographic hash function, there is only a negligible chance of collisions. In the following pseudocode excerpts, we say that council member c_i has address i and simplify shadows from $u_{j\to i} = \langle x_i, y_{j\to i} = f_j(x_i)\rangle$ to $u_{j\to i} = f_j(i)$ and likewise for shares. We also use the addresses of council members to index data structures in Figure 4.4, like the public keys used for shadow encryption and decryption `pks[]`. For persistently storing received

```
Prepare register():
    Draw $\mathrm{sk}_i$ uniformly at random from $\mathbb{Z}_p$ and store it persistently
    Generate $\mathrm{pk}_i = g^{\mathrm{sk}_i}$
    Set $\mathsf{members} \longleftarrow \mathsf{members} \cup i$
    Send transaction register($\mathrm{pk}_i$)
Receive register($\mathrm{pk}_j$) from $c_j$:
    Set $\mathsf{pks}[j] \longleftarrow \mathrm{pk}_j$
    Set $\mathsf{members} \longleftarrow \mathsf{members} \cup j$
```

Figure 4.5.: Pseudocode for preparing and receiving registrations to ETHTID contract instances.

values in these data structures, we use arrows, for example $\mathsf{pks}[j] \longleftarrow \mathrm{pk}_j$, whereas we use equal signs in the computation and labeling of temporary variables.

During *initialization*, an initiator deploys an ETHTID smart contract through a transaction submitted to the Ethereum peer-to-peer network. In our implementation, the threshold t is codified as a fraction of the number of participants that register in the next phase. Additionally, the initiator of a particular ETHTID instance also defines the schedule of all subsequent phases relative to the block that contains the deployment transaction. In a practical setting, an ETHTID instance would also need to be supplied with a reward during or shortly after initialization in order for prospective council members to decide on their participation.

In the *registration* phase, council members register with a particular ETHTID smart contract instance and submit a newly generated ephemeral public key $\mathrm{pk}_i := g^{\mathrm{sk}_i}$ to be stored within the contract while keeping the corresponding secret key sk_i to themselves, as depicted in Figure 4.5. These key pairs are only used for the exchange of encrypted shadows between council members via a particular ETHTID smart contract instance and discarded afterwards. The reason for exchanging encrypted shadows in this way rather than letting council members do so confidentially off-chain is to enable the non-interactive dispute resolution that we describe below. In practice, volunteers would also submit a required deposit with their registration which can be destroyed or redistributed in the event they provably misbehave. For our proof of concept implementation, we used a simple first-come-first-serve[8] registration mechanism, which is very much vulnerable to Sybil attacks [27], the ramifications of which we examine in more detail in Subsection 4.6.2. Once the registration phase ends based on block height, the size of the council n, the Ethereum addresses of its members, and the threshold t as a fraction of n are fixed.

With all necessary parameters settled, council members can proceed to *distribute shadows* as depicted in Figure 4.6. Please recall from Section 4.3 that each council member c_i draws a random polynomial $f_i(x) = d_i + \sum_{k=1}^{t} a_{i,k}x^k$ containing as its y-intercept their contribution to the decryption key. Based on this polynomial, member c_i generates $e_i = g^{d_i}$,

[8] Contrary to the common use of this phrase, participants register to serve as council members, hence "serve" instead of "served".

```
Prepare distribute_shadows():
    Obtain t from ETHTID contract
    Draw $d_i$ uniformly at random from $\mathbb{Z}_p$
    Generate $e_i = g^{d_i}$
    Set es[$i$] $\leftarrow e_i$
    Draw $\{a_{i,k}\}_{k=1}^{t}$ uniformly at random from $\mathbb{Z}_p$
    Generate $\{A_{i,k} = g^{a_{i,k}}\}_{k=1}^{t}$
    Set poly_comms[$i$] $\leftarrow \{A_{i,k}\}_{k=1}^{t}$
    for $l \in$ members do // Generate shadows for other members and self
        Generate $u_{i \to l} = d_i + \sum_{k=1}^{t} a_{i,k} l^k$
        if $l = i$ then // Shadow for self
            Set dec_shadows[$i$] $\leftarrow u_{i \to l}$
        else // Shadow for other member
            Load $\mathrm{pk}_l \leftarrow$ pks[$l$]
            Generate $k_{i,l} = \mathrm{pk}_l^{\mathrm{sk}_i}$
            Generate $\overline{u_{i \to l}} = u_{i \to l} \oplus \mathrm{H}(k_{i,l} \parallel l)$
        end
    end
    Send transaction distribute_shadows($\{\overline{u_{i \to j}}\}_{j \in \mathsf{members} \setminus i}, e_i, \{A_{i,k}\}_{k=1}^{t}$)
Receive distribute_shadows($\{\overline{u_{j \to l}}\}_{l \in \mathsf{members} \setminus j}, e_j, \{A_{j,k}\}_{k=1}^{t}$) from $c_j$:
    Set enc_shadows[$j$] $\leftarrow \{\overline{u_{j \to l}}\}_{l \in \mathsf{members} \setminus j}$
    Set es[$j$] $\leftarrow e_j$
    Set poly_comms[$j$] $\leftarrow \{A_{j,k}\}_{k=1}^{t}$
    Generate $k_{i,j} = \mathrm{pk}_j^{\mathrm{sk}_i}$
    Generate $u_{j \to i} = \overline{u_{j \to i}} \oplus \mathrm{H}(k_{i,j} \parallel i)$ // Decrypt shadow for self
    if $g^{u_{j \to i}} \neq e_j \prod_{k=1}^{t} A_{j,k}^{i^k}$ then // Shadow invalid
        Call submit_dispute against $c_j$ // See Figure 4.7
    else // Shadow valid
        Set dec_shadows[$j$] $\leftarrow u_{j \to i}$
    end
```

Figure 4.6.: Pseudocode for preparing and receiving shadow distribution broadcasts in ETHTID.

which acts both as a commitment to d_i and as their contribution to the encryption key, and $\{A_{i,k} = g^{a_{i,k}}\}_{k=1}^{t}$, the remaining commitments to their polynomial used for shadow verification. Next, member c_i generates shadows $\{u_{i \to j} = f_i(j)\}_{j \in \mathsf{members}}$, including one for themselves. Please recall that j above is the registered Ethereum address of member c_j interpreted as an integer. The dispute resolution function that we explain in the next phase enforces adherence to this principle by all members as any shadows not generated in this way are provably invalid and cause for disqualification. By reducing shadows to just their y-coordinate, since their x-coordinate is public knowledge, the costs for their inclusion in transactions is reduced significantly.

```
Prepare submit_dispute():
    Load $\mathrm{pk}_j \leftarrow \mathsf{pks}[j]$
    Generate $k_{i,j} = \mathrm{pk}_j^{\mathrm{sk}_i}$
    Generate $\pi(k_{i,j})$ via Figure 4.8
    Load $\{\overline{u_{j\to l}}\}_{l\in\mathsf{members}\setminus j} \leftarrow \mathsf{enc\_shadows}[j]$
    Load $e_j \leftarrow \mathsf{es}[j]$
    Load $\{A_{j,k}\}_{k=1}^{t}, k_{i,j} \leftarrow \mathsf{poly\_comms}[j]$
    Send transaction submit_dispute($\{\overline{u_{j\to l}}\}_{l\in\mathsf{members}\setminus j}, e_j, \{A_{j,k}\}_{k=1}^{t}, k_{i,j}, \pi(k_{i,j})$)
Receive submit_dispute($\{\overline{u_{j\to l}}\}_{l\in\mathsf{members}\setminus j}, e_j, \{A_{j,k}\}_{k=1}^{t}, k_{m,j}, \pi(k_{m,j})$) against $c_j$ by $c_m$:
    // ETHTID contract adjudicates validity of dispute, members process a valid
       dispute as follows
    Set members $\leftarrow$ members $\setminus j$
    Delete pks[$j$]
    Delete enc_shadows[$j$]
    Delete dec_shadows[$j$]
    Delete es[$j$]
    Delete poly_comms[$j$]
```

Figure 4.7.: Pseudocode for preparing and receiving disputes in ETHTID.

Each member encrypts shadows for other members as follows. Based on the ephemeral public keys registered in the ETHTID smart contract, both members c_i and c_j derive a shared secret via Diffie-Hellman(-Merkle) key exchange [26, 59]: $k_{i,j} = \mathrm{pk}_j^{\mathrm{sk}_i} = \mathrm{pk}_i^{\mathrm{sk}_j} = k_{j,i}$. Since Ethereum does not provide any symmetric encryption primitives, the authors of EthDKG opted to use a one-time pad encryption, which we adopt without changes and recall here for completeness. As reusing the same pad leaks some information about the plaintext, c_i and c_j derive unique pads from their shared secret through a cryptographic hash function before encrypting the shadow for the other member:

$$\overline{u_{i\to j}} = u_{i\to j} \oplus \mathrm{H}(k_{i,j} \parallel j).$$

Decryption functions analogously by the recipient creating the same pad and applying it with the XOR operator to the encrypted shadow. This construction exploits the fact that both the y-coordinate of shadows and the output of the Keccak hash function in Ethereum, H() above, are 256 bit values. Lastly, each council member c_i distributes both encrypted shadows and polynomial commitments in a transaction that calls `distribute_shadows` of the ETHTID smart contract instance. Rather than storing all submitted values in the contract state, which would be rather expensive, only the contribution to the encryption key e_i and one hash over both encrypted shadows and commitments is stored. While the contribution e_i is necessary to derive the full encryption key e on-chain, the hash over the broadcast values is necessary for the contract to verify their integrity when they are resubmitted as part of a dispute. All submitted values are also emitted in a `ShadowDistribution` event for easier retrieval by other council members.

Each council member c_i retrieves the values broadcast by all other members, decrypts shadows intended for themselves as described above, and verifies their validity against

```
Algorithm 1: DLEQ($g, \mathrm{pk}_i, \mathrm{pk}_j, k_{i,j}, \mathrm{sk}_i$)
To show that $\mathrm{dlog}_g(\mathrm{pk}_i) = \mathrm{dlog}_{\mathrm{pk}_j}(k_{i,j})$ holds without revealing the discrete logarithm $\mathrm{sk}_i$, a
  prover proceeds as follows
Draw $w$ uniformly at random from $\mathbb{Z}_p$
Compute $t_1 = g^w$ and $t_2 = \mathrm{pk}_j^w$
Compute $c = \mathrm{H}(g \parallel \mathrm{pk}_i \parallel \mathrm{pk}_j \parallel k_{i,j} \parallel t_1 \parallel t_2)$
Compute $r = w - \mathrm{sk}_i c \pmod p$
Output $\pi(k_{i,j}) = \langle c, r \rangle$
```

Figure 4.8.: Non-interactive zero-knowledge proof of correctness for symmetrical encryption keys.

```
Algorithm 2: DLEQ-Verify($g, \mathrm{pk}_i, \mathrm{pk}_j, k_{i,j}, \pi(k_{i,j})$)
To check the correctness of a proof $\pi(k_{i,j}) = \langle c, r \rangle$, showing that $\mathrm{dlog}_g(\mathrm{pk}_i) = \mathrm{dlog}_{\mathrm{pk}_j}(k_{i,j})$ holds,
  i.e. that $k_{i,j}$ is correct, a verifier proceeds as follows
Compute $t'_1 = g^r \cdot \mathrm{pk}_i^c$ and $t'_2 = \mathrm{pk}_j^r \cdot k_{i,j}^c$
Output VALID if $c = \mathrm{H}(g \parallel \mathrm{pk}_i \parallel \mathrm{pk}_j \parallel k_{i,j} \parallel t'_1 \parallel t'_2)$
Output INVALID otherwise
```

Figure 4.9.: Verification procedure for the proof in Figure 4.8

the polynomial commitments of its sender via Equation (4.2) as depicted in the second half of Figure 4.6. In the event that member c_i obtained an incorrect share from member c_j, c_i files a *dispute* by submitting a transaction calling the `submit_dispute` function of the ETHTID contract, which is depicted in Figure 4.7. As part of this transaction, the issuing member c_i resubmits the encrypted shadows and polynomial commitment values that the accused member c_j broadcast during shadow distribution so that the contract can verify the shadow in question. The integrity of this replayed broadcast is verified by the contract via the previously stored hash. In order for the contract to decrypt the shadow $\overline{u_{j \to i}}$, member c_i must also submit the shared key $k_{i,j}$ and prove its correctness without revealing their ephemeral private key sk_i. Without such a correctness proof, it would be trivial to disqualify any honest member through unfounded disputes by including an incorrect shared key, which in turn would make the decrypted shadow look invalid. Broadcasting the ephemeral secret key sk_i directly is also not advisable as that would immediately reveal all shadows generated by member c_i, which in turn can be used to recover their polynomial and ultimately their contribution to the decryption key d_i, effectively disqualifying themselves. We adopt from Schindler *et al.* [79] a non-interactive zero-knowledge proof and verification scheme derived from the works of Chaum and Pedersen [21] and Camenisch and Stadler [18], which we introduce here for completeness.

Please recall that the shared key $k_{i,j}$ is generated by member c_i as $\mathrm{pk}_j^{\mathrm{sk}_i}$ and that both the ephemeral public keys pk_i and pk_j have been submitted to and stored in the ETHTID contract instance that a dispute is submitted to. To prove the correctness of the shared key $k_{i,j}$ thus reduces to proving the equality between two discrete logarithms: $\mathrm{dlog}_g(\mathrm{pk}_i)$

and $\text{dlog}_{\text{pk}_j}(k_{i,j})$, which are both sk_i, without revealing it. In an *interactive* proof, a prover would commit to a random value w via the values g^w and pk_j^w, the verifier would supply a challenge c, and the prover would then complete the proof by calculating $r = w - c\text{sk}_i$. To verify the proof, a verifier would check to see if $g^r \cdot \text{pk}_i^c = g^w$ and if $\text{pk}_j^r \cdot k_{i,j}^c = \text{pk}_j^w$. The ability to compute r such that both of these checks hold proves knowledge of sk_i but more importantly that both discrete logarithms are equal. Note that the prover has to commit to w, the result in the exponent, before knowing the challenge c. It is this logical dependence that unequivocally proves knowledge of sk_i as there is no efficient way to compute a suitable response r otherwise. Such an interactive proof can be made *non-interactive* via the Fiat-Shamir heuristic [32]: By replacing the verifier-supplied challenge c with a value derived from instance-specific data via a cryptographic hash function $\text{H}()$, including it in the supplied proof, and adjusting the verification accordingly, the zero-knowledge proof and thus the entire dispute handling can be completed in a single transaction. Note that the aforementioned dependence of a prover committing to w before knowing the challenge c remains in the non-interactive setting. Figures 4.8 and 4.9 depict this non-interactive zero-knowledge proof generation and verification concisely.

With the resubmitted broadcast of the accused member c_j and the shared key $k_{i,j}$ verified, the ETHTID contract can decrypt and verify the shadow in question via Equation (4.2). If the shadow is indeed incongruent with the corresponding polynomial commitments, member c_j is disqualified from the remaining procedure and their contribution e_j is discarded. Other members observe the contract-defined outcome of a dispute and if the dispute is valid, i.e. member c_j distributed an invalid shadow, discard all information they received from member c_j as depicted in the second half of Figure 4.7. With an incentive scheme in place, member c_j would also lose the deposit they submitted during registration. A dispute transaction is aborted and no state changes are persisted in case any of the above checks fail. Once the dispute phase ends, based on elapsed blocks, it can be assumed that the distributed key generation and threshold sharing was concluded correctly and that any $t + 1$ council members can recover the shared decryption key d with their shares.

The *submission* phase begins with a council member calling the function `generate_e` through a transaction, during which the ETHTID contract instance combines the contributions e_i of the remaining, qualified council members into the group encryption key e. The generation of e is also documented through the `EK` event on the Ethereum blockchain. With the encryption key e, users of a given ETHTID instance can prepare their messages to be disclosed in a coordinated way at time $\text{t}^{(\text{d})}$. In Chapter 5, we describe this process in more detail using a Diffie-Hellman(-Merkle) key exchange whereas we initially envisioned the use of ElGamal encryption [86]. Depending on the schedule set by the initiator in the beginning, this phase can last for days, weeks, or months. Similar to the previous phases, the end of this phase is specified via the height of the Ethereum blockchain. Due to the variance in block creation time in Ethereum, the more blocks are specified to pass in this phase, the more uncertain the time for disclosure becomes in wall clock time.

Once the time for disclosure $\text{t}^{(\text{d})}$, specified as height of the Ethereum blockchain, has arrived, qualified council members are encouraged to recover and publish the decryption key d as depicted in Figure 4.10. To facilitate recovery, they can submit their shares

```
Prepare distribute_share():
    Generate $r_i = \sum_{j \in \text{members}} \text{dec\_shadows}[j]$
    Set $\text{shares}[i] \leftarrow r_i$
    Send transaction distribute_share($r_i$)
Receive distribute_share($r_j$) from $c_j$:
    Generate $e = \prod_{l \in \text{members}} \text{es}[l]$ or obtain $e$ from ETHTID contract
    for $k = 1$ to $t$ do // Combine polynomial commitments
        Generate $A_k = \prod_{l \in \text{members}} A_{l,k}$ // From poly_comms[$l$]
    end
    if $g^{r_j} = e \prod_{k=1}^{t} A_k^{j^k}$ then // Share is valid
        Set $\text{shares}[j] \leftarrow r_j$
        if $|\text{shares}| = t + 1$ then // Enough valid shares for recovery
            Compute $d$ via Equation (4.3)
            Send transaction submit_d($d$)
        end
    end
```

Figure 4.10.: Pseudocode for preparing and receiving share distribution broadcasts in ETHTID.

r_i via a helper transaction to `distribute_share` which merely broadcasts the share in a `ShareDistribution` event and does not affect the ETHTID contract state in any way. Note that `ShareDistribution` does not include a validity check for the submitted share as that would be both expensive and pointless at this stage of the protocol. Unlike the `distribute_shadows` transaction, the use of `distribute_share` is technically optional as council members can publish or exchange their shares in any other way so long as one member can recover and publish the decryption key d in a timely manner to release the council as a whole from its contractual obligation. The correctness of shares r_i can be verified similar to shadows as depicted in the second part of Figure 4.10. Please recall from Section 4.3 that the decryption key d is the y-intercept of the group polynomial which is defined as the sum of the polynomials of qualified members. Just like how contributions e_i were combined to form the encryption key e, so too can the polynomial commitments $\{A_{i,k}\}_{k=1}^{t}$ of all qualified members be combined into commitments $\{A_k = \prod_{i=1}^{n} A_{i,k}\}_{k=1}^{t}$ to the group polynomial. Then, these commitments can be used to verify the correctness of shares with Equation (4.2). Let $R \subset \text{members}$ with $|R| = t + 1$ be the council members participating in the recovery of d by publishing valid shares, the decryption key can then be recovered via Lagrange interpolation [89] as:

$$d = \sum_{k \in R} \left(\prod_{l \in R \setminus k} \frac{l}{l - k} r_k \right). \tag{4.3}$$

Please recall that the indices above are stand-ins for the Ethereum addresses of council members interpreted as integers. One council member can then submit the recovered decryption key d to the ETHTID smart contract instance via a transaction to the `submit_d` function, during which the contract verifies that $g^d = e$. Once the correct decryption key d

has been submitted, the council has fulfilled its contractual obligation and the contents of all appropriately prepared messages become available to anyone. Similar to the generation of the encryption key, the release of the decryption key is also recorded with a `DK` event on the Ethereum blockchain, unequivocally ending the submission phase.

4.5.3. Optimizations

In the above description of ETHTID, we pointed out aspects of EthDKG that we adopted. In this section, we describe in more detail *what* we changed compared to EthDKG in order to save costs and explain *why* these changes are both sensible and safe. We provide quantitative measurements on the cost savings in Subsection 4.6.1.

The first optimization is rather straightforward but noteworthy nonetheless. The Barreto-Naehrig curve that Ethereum supports is a so called “pairing-friendly” bilinear elliptic curve, a feature that EthDKG makes use of in its distributed generation of a BLS signature key pair [15, 14]. Since ETHTID only requires a discrete log key pair, we were able to forgo any operation that relies on the bilinearity property of the BN curve, including pairing checks. The reason we still use the BN curve is that there is currently no alternative available on Ethereum. EIP-1829[9] or EIP-1962[10] would potentially allow us to use elliptic curves that are better suited to our application, such as secp256k1[11] or curve25519[12], but they have not seen much activity in recent years. However, with Ethereum’s switch to Proof of Stake completed in September 2022, such functional improvements to support new or improve existing applications may garner more attention. With ETHTID, we add to the rationale for including operations on elliptic curves besides the existing BN curve.

The second optimization we made revolves around biasing of generated key pairs and our deliberate choice of accepting such attacks rather than defending against them. Gennaro *et al.* [36] demonstrate that many DKG procedures for discrete-log-based cryptosystems, including the Joint-Feldman VSS we use, are vulnerable to biasing of the shared secret by an attacker. In our protocol, such an attack would be performed by an attacker controlling or posing as multiple, but not more than t, council members and selectively disqualifying some of them to bias the resulting key pair. These biasing attacks are possible because an attacker can simulate what the decryption key would look like based on which of their contributions are included before the dispute phase ends. EthDKG defends against this kind of attack through a mechanism by Neji *et al.* [65], which necessitates an additional broadcast by all participants and reconstruction of key contributions of parties that remain inactive after the dispute phase. We deliberately chose to accept the risk of adversarial council members biasing the key pair during the DKG setup in favor of saving significant costs for the following three reasons:

[9] `https://eips.ethereum.org/EIPS/eip-1829`
[10] `https://eips.ethereum.org/EIPS/eip-1962`
[11] `http://www.secg.org/sec2-v2.pdf`
[12] `https://www.rfc-editor.org/rfc/rfc7748`

- **Security Foundation**: The security of our construction is based on the difficulty of computing discrete logarithms and not the uniformly random distribution of the used key pair. In an updated version of their original paper, Gennaro *et al.* [35] show that in such cases, a biasing attacker gains no significant advantage as computing discrete logarithms remains difficult even with a limited degree of biasing.
- **Attacker Costs**: To participate with more than one identity in an ETHTID execution in order to execute a biasing attack, an attacker would have to bear additional costs and, depending on the incentive scheme used, be willing to lose security deposits required during registration.
- **Limited Lifetime**: By construction, each key pair generated in an ETHTID execution has a limited lifetime before the decryption key is published. This leaves an attacker only little time to take advantage of a bias they added to a given key pair.

In our original paper [87, 86], we also showed that biasing attacks on an ETHTID key pair are unnecessary if said key pair is used for ElGamal encryption because such ciphertexts can be biased afterwards. In Chapter 5, we end up using a Diffie-Hellman(-Merkle) key exchange rather than ElGamal encryption of to-be-disclosed messages, but we reproduce our security argument here for completeness:

Given a discrete log key pair $d, e = g^d$, an ElGamal encryption of a message $m \in \mathbb{Z}_p$ is constructed by sampling a random element $r \in \mathbb{Z}_p$ and computing $(c_1, c_2) = (g^r, e^r \cdot m)$. Now assume that, during a distributed generation of the above key pair, an attacker waits and observes all e_i from honest parties, computing an unbiased encryption key $\tilde{e} = \prod e_i$. The attacker may then choose any bias b and force the resulting encryption key to be $e = \tilde{e} \cdot g^b$. Whatever attack an adversary can perform on ciphertexts encrypted with the biased key e can also be performed against ciphertexts created with the unbiased key $\tilde{e}$. Given $\tilde{e}$ and a cipher text $(\tilde{c}_1, \tilde{c}_2) = (g^r, \tilde{e}^r \cdot m)$, and given b, this cipher text can be transformed to the biased encryption key e: $(c_1, c_2) = (\tilde{c}_1, \tilde{c}_2 \cdot \tilde{c}_1^b) = (g^r, \tilde{e}^r \cdot g^{rb} \cdot m) = (g^r, (\tilde{e} \cdot g^b)^r \cdot m)$ which is a valid cipher text under $e = \tilde{e} \cdot g^b$.

Consequently, an attacker gains no advantage by introducing a bias in the distributed key generation process because any ElGamal ciphertexts created with an unbiased key pair can simply be biased afterwards as well.

4.6. Evaluation

Before evaluating our implementation of ETHTID, it is instructive to briefly examine the complexity of our concept. As mentioned previously, ETHTID proceeds in six phases as depicted in Figure 4.3, but only two of those phases are broadcast rounds where each council member is supposed to send a transaction: registration and shadow distribution. Looking at these two phases more closely, the registration transaction is of constant size as it only contains one ephemeral public key regardless of council size or threshold. However, the shadow distribution transaction does scale with both of those parameters as it contains

Table 4.3.: Costs of functions independent of threshold t and council size n. Conversion of gas to USD via daily average exchange rates for 1st June 2022 as reported by Etherscan: USD 1817.42 per ETH, ETH 60.06×10^{-9} per gas.

Function	Gas	USD
Contract Deployment	1 879 437	205.13
`register`	113 914	12.43
`distribute_share`	25 834	2.82
`submit_d`	56 729	6.19

$n - 1$ encrypted shadows and $t + 1$ polynomial commitment values[13]. Disputes, meanwhile, are a single transaction by only one member that inherits its scaling behavior from the shadow distribution transaction as it resubmits encrypted shadows and polynomial commitments. Lastly, for a successful recovery only t share submission transactions are necessary, either via the Ethereum blockchain or another communication channel, followed by one transaction by a member who has not yet submitted their share to submit d. Overall, the message complexity of ETHTID as a protocol scales quadratically with the number of council members.

Like Palinodia in the last chapter, we implemented ETHTID[14] as an Ethereum smart contract in Solidity 0.8 and deployed it to a local development blockchain on the London hard fork using Ganache v7.2.0 of the Truffle Suite development tools.

We first provide a gas cost evaluation before examining ETHTID's security properties and their limits in more detail.

4.6.1. Gas Costs & Performance

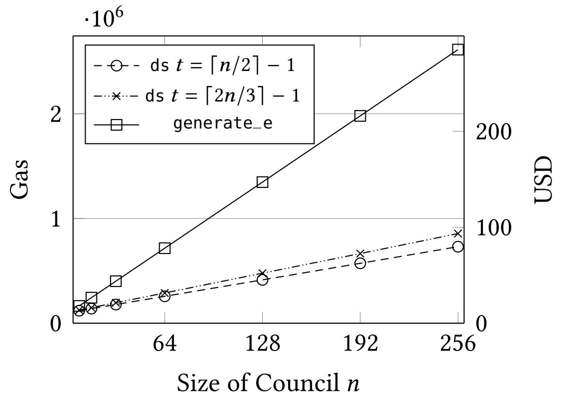
We measured gas costs for contract deployment as well as all mandatory and optional transactions for a full execution of the coordinated disclosure protocol with varying council size n and thresholds t. In particular, we examined the "simple majority" threshold $t = \lceil n/2 \rceil - 1$ and the "supermajority" case of $t = \lceil 2n/3 \rceil - 1$. To automate the gas cost evaluation, we adopted and expanded the Python application of EthDKG that Schindler *et al.* [79] kindly made public along with their Solidity implementation of EthDKG. As we observed very regular and predictable costs, we capped our evaluation at $n = 256$. For better intuition, we again report costs in both units of gas and in USD based on the daily average exchange rates for 1st June 2022 as reported by Etherscan[15].

In Table 4.3, we provide costs that are independent of council size and threshold. While both `distribute_share` and `submit_d` are essentially a broadcast of a single 256 bit value, their difference in execution cost stems from two additional operations in `submit_d`: First,

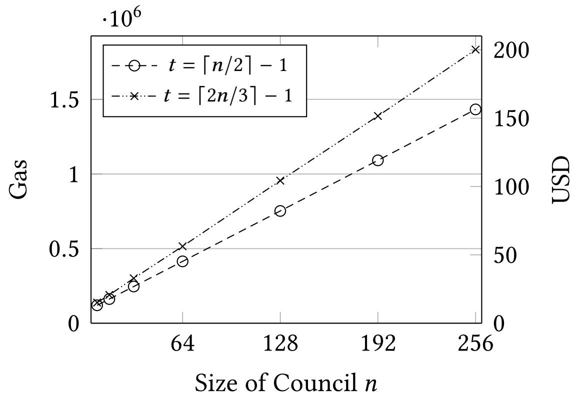
[13] Please recall that the contribution e_i is also a commitment to the constant part of $f_i(x)$

[14] `https://git.scc.kit.edu/dsn-projects/dissertations/dsim/-/tree/main/ETHTID`

[15] `https://etherscan.io/`



(a) Costs of `distribute_shadows` (ds) for thresholds $t = \lceil n/2 \rceil - 1$ and $t = \lceil 2n/3 \rceil - 1$, and `generate_e`, which is independent of t.



(b) Costs of `submit_dispute` for thresholds $t = \lceil n/2 \rceil - 1$ and $t = \lceil 2n/3 \rceil - 1$.

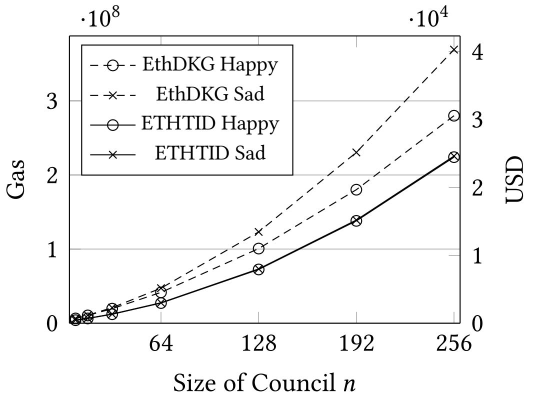
Figure 4.11.: Conversion of gas to USD via daily average exchange rates for 1st June 2022 as reported by Etherscan: USD 1817.42 per ETH, ETH 60.06×10^{-9} per gas. [87]

Table 4.4.: Best fit curves for scaling costs in ETHTID based on linear and quadratic regression.

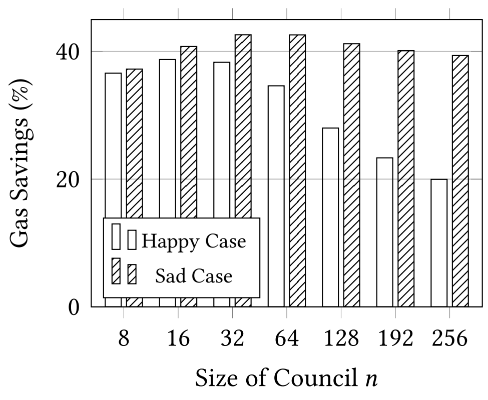
Function	Best Fit Curve
`distribute_shadows` ($t = \lceil n/2 \rceil - 1$)	$2462n + 99\,664$
`distribute_shadows` ($t = \lceil 2n/3 \rceil - 1$)	$2944n + 101\,121$
`generate_e`	$9866n + 85\,518$
`submit_dispute` ($t = \lceil n/2 \rceil - 1$)	$5291n + 76\,745$
`submit_dispute` ($t = \lceil 2n/3 \rceil - 1$)	$6832n + 81\,042$
Total (happy case)	$2474n^2 + 233\,696n + 2\,076\,171$
Total (sad case)	$2474n^2 + 239\,013n + 2\,155\,784$

a check is performed to ensure that the submitted decryption key is consistent with the encryption key generated during `generate_e`. Second, the submitted decryption key is stored as part of the contract state to ensure its long-term availability whereas shares submitted during `distribute_share` are merely emitted as part of a `ShareDistribution` event.

With Figure 4.11, we show the execution costs of operations that scale with council size and threshold. Figure 4.11a shows the costs for shadow distribution and encryption key generation. Note that the broadcast as part of `distribute_shadows` consists of $n - 1$ encrypted shadows and $t + 1$ values for the polynomial commitment, thus leading to a scaling behavior in both variables. While only e_i, i.e. the first value in a polynomial commitment, along with a hash over both encrypted shadows and commitments is stored as part of the contract state, each byte submitted as part of a transaction incurs a small gas cost. Additionally, the gas price for the hashing operation scales with the size of its input, which also depends on the council size n and threshold t. The execution costs for `generate_e` only scale linearly with council size, as that parameter determines the number of contributions e_i that must be combined during this transaction. Figure 4.11b meanwhile



(a) Total execution costs of EthDKG and ETHTID with $t = \lceil n/2 \rceil - 1$.



(b) Relative cost savings achieved by ETHTID compared to EthDKG.

Figure 4.12.: Happy case: No misbehavior, all members participate honestly. Sad case: One incorrect shadow distribution and subsequent dispute and only $t + 1$ council members remain to complete each protocol. Conversion of gas to USD via daily average exchange rates for 1st June 2022 as reported by Etherscan: USD 1817.42 per ETH, ETH 60.06×10^{-9} per gas. [87]

displays the costs for `submit_dispute` where none of the checks described in Section 4.5 fail and cause an early abort. The majority of costs for filing valid disputes arises from the evaluation of Equation (4.2), which scales with the threshold t as it determines the degree of the polynomial. Since the values broadcast during `distribute_shadows` by an offending member are resubmitted as part of `submit_dispute`, the council size has a small influence as well. One pair of executions that demonstrates this influence very neatly are $n = 192, t = \lceil 2n/3 \rceil - 1 = 127$ with 1 388 745 gas and $n = 256, t = \lceil n/2 \rceil - 1 = 127$ with 1 432 846 gas, since they have the same threshold but different council sizes. During the execution of `submit_dispute`, the additional 64 encrypted shadows lead to additional costs of 44 101 gas or USD 4.81 with the exchange rates quoted in Figure 4.11. In Table 4.4, we provide linear and quadratic fits of the scaling costs displayed in Figures 4.11 and 4.12a.

To demonstrate the efficacy of our optimizations as outlined in Subsection 4.5.3, we performed a direct comparison of the overall execution costs of EthDKG and ETHTID and depict the results in Figure 4.12. We examine two scenarios for both applications: a happy case where no participant misbehaves and only the minimally necessary amount of transactions are executed, and a sad case where one participant distributes invalid shadows and necessitates the filing of a dispute and only the minimally necessary number of participants remain active to complete each protocol. For the purpose of this comparison, we repeated the automated gas cost evaluation of EthDKG on the London hard fork after minimally updating the EthDKG code to comply with Solidity version 0.8. In the happy case, the overall costs for ETHTID are comprised of one contract deployment, n calls of `register` and `distribute_shadows` each, one call of `generate_e`, $t + 1$ broadcasts via `distribute_share`, and finally one call of `submit_d`. For the sad case, one execution of `submit_dispute` is added. Note that EthDKG does not include the functionality to recover and publish the distributedly generated group secret. However, based on our measurements from Table 4.3, the overhead for adding such functionality would be rather small in comparison to the other costs. The EthDKG equivalent of `submit_d` would also be slightly more expensive as a pairing check would be required during the validity check of the submitted key. The plot in Figure 4.12a very clearly shows the cost impact of the biasing defenses in EthDKG that we deliberately omitted in ETHTID. Additionally, the discrepancy between happy and sad case for EthDKG and ETHTID also shows the potential cost impact that inactive or misbehaving participants can have due to these defenses. Overall, we observe gas savings of 20–40 % depending on council size and scenario as depicted in Figure 4.12b.

At the time of writing, blocks in Ethereum have a limit of $30{\times}10^6$ gas that can be consumed by transactions, although the protocol incentivizes miners to keep blocks half-full and reserve the remaining capacity to feather short-term spikes in transaction demand. As such, all operations we evaluated above fall well below this limit and are thus technically feasible, albeit financially challenging, to execute. The first operation that would run into the soft $15{\times}10^6$ gas limit would be `submit_dispute`. Using the corresponding linear fit curve in Table 4.4 with $t = \lceil n/2 \rceil - 1$ yields a limit of $n = 2820$. Plugging $n = 2820$ into the quadratic fit curve for the total happy case costs in Table 4.4 suggests overall costs of $20.3{\times}10^9$ gas or USD $2.2{\times}10^6$ with the exchange rates quoted in Figure 4.12.

4.6.2. Security Considerations

Unlike Palinodia, ETHTID derives the desired properties we stated in Section 4.1 in parts from its underlying ledger, from its cryptographic construction, and from the behavior of each council. In this section, we clarify these relations and their limits.

Fairness: ETHTID achieves the fairness property by construction and through the availability guarantees of its underlying ledger. Since the contents of all appropriately prepared messages are dependent on the availability of the decryption key, which either is or is not persistently stored as part of an ETHTID contract instance's state, either all message contents are available or none are. Furthermore, the availability of ledger data ensures that, if the decryption key is published via a transaction, it is available to everyone eventually. An attacker seeking to hide a published decryption key from a particular user would have to either prevent them from accessing ledger data indefinitely, which is rather obvious to said user, or produce a convincing alternate ledger where the decryption key was never published, which would break the fundamental DLT assumptions we stated in Section 4.4. Lastly, the availability of committed messages is also relevant to the fairness property. We consider this aspect more prevalent to applications using ETHTID and examine it further in the next chapter.

Hiding: The hiding property in ETHTID is achieved through both its cryptographic construction as well as the behavior of an instance's council. Through the use of distributed key generation and threshold secret sharing coupled with the assumption that no more than t council members are Sybil identities of an attacker or compromised, the decryption key is unavailable to any party before its scheduled recovery. Consequently, the contents of all appropriately prepared messages remains known only to their authors until disclosure. To break the hiding property, an attacker would have to obtain the decryption key d

before the time of disclosure. Computing $\text{dlog}_g\, e$ is precluded by the assumption that discrete logarithms are difficult to compute for the underlying cyclic finite group, which applies to the Barreto Naehrig curve we use, and the computational bound we assume for attackers. A more approachable attack avenue is to compromise or bribe sufficiently many council members to obtain $t + 1$ valid shares prematurely and privately recover the decryption key d, which violates our assumption that at most t members of a council are corrupt or compromised. At first glance, it would seem compelling to equip ETHTID smart contracts with the ability to punish council members if anyone else provided information that was supposed to remain secret, thereby making the first and necessary step towards such a collusion a significant risk. However, due to the core functionality of threshold secret sharing, assigning blame for such information leaks with certainty is not possible. Please recall from Figure 4.1 the operations marked with dotted arrows that were possible but not necessary for Joint-Feldman VSS. In particular, $t + 1$ members can pool shadows they received from member c_i to recover their polynomial $f_i(x)$ and use it to recreate shadows for any other council member c_j. Similarly, $t + 1$ members can recover the group polynomial $f(x)$ not only to recover the decryption key d but also to generate shares r_i of other council members. Consequently, any information that council members are supposed to keep secret, besides the ephemeral secret key sk_i, can become public through no fault of their own, making a punishment mechanism as sketched above more of a liability than a useful feature.

Binding: ETHTID on its own does not ensure the binding property as it mostly depends on the mechanism by which messages are committed and stored until and beyond disclosure, i.e. implementation details of an application using ETHTID for coordinated disclosure. Similar to the message availability mentioned above for the fairness property, we examine this aspect in more detail in the next chapter. To give a brief preview, by logging integrity-protecting references to encrypted messages on-chain and disseminating them through a decentralized file sharing protocol like the InterPlanetary File System (IPFS), the binding property can be achieved. In this case, the immutability of committed messages is derived from the immutability and availability properties of the ledger used to log the references, whereas the availability of messages is derived from the functional properties of the employed distribution and storage system.

4.7. Discussion

With ETHTID, we demonstrate that orchestrating and documenting a coordinated disclosure is possible on smart-contract-capable public ledgers that support the necessary elliptic curve operations. By carefully analyzing the requirements of our use case, we are able to significantly lower costs compared to previous works like EthDKG. While we developed ETHTID for our particular use case of enabling decentralized software reviews, which is the main focus of the next chapter, we also believe that it has independent value to facilitate or simplify other processes with similar requirements, like sealed-bid auctions, appraisal of digital or physical objects, or the enforcement of press embargoes. Note that the construction of ETHTID places no restrictions on the size or type of messages that can be disclosed. Through Diffie-Hellman(-Merkle) key exchange or by encrypting a symmetric key via ElGamal encryption, arbitrarily large files can be prepared for a coordinated disclosure. Also note that, while the public availability of encryption and decryption keys is essential in ETHTID, the to-be-disclosed messages do not have to be public. As such, public ETHTID instances can also be used to disclose messages to a limited number of parties by restricting the dissemination of prepared messages.

Since the availability of both encryption and decryption keys is not restricted in ETHTID, multiple applications can use and rely on a single instance with a suitable schedule. However, this may be a double-edged sword in practice when a council's adherence to said schedule is primarily motivated through incentives. From a game-theoretic point of view, applications that attach themselves to an existing ETHTID instance should contribute to its reward pool to inoculate its council against bribery, which becomes increasingly more enticing the more potentially valuable information is governed by said instance. This dynamic has the potential to result in a tragedy of the commons where a publicly available resource, i.e. an ETHTID instance, is rendered useless through overuse.

On the topic of economic concepts, ETHTID in conjunction with the previously mentioned work by Yakira *et al.* [91] is a concrete example of a notion that has only become practical with decentralized ledgers supporting smart contracts: the intertwining of cryptographic protocols with economic rewards. In particular, DLTs enable "credible threats and promises", a fundamental concept of economic game theory: If conditions that a smart contract can verify are met, consequences can be enforced as specified in the contract's code. In the case of ETHTID, this mostly concerns the functionality of filing disputes against council members that distribute invalid shadows. Curiously enough, the fact that disputes *can* be filed and consequences *can* be enforced should lead to this functionality being rarely used in practice, if ever. The dispute mechanism acts as a deterrent since all potential and actual participants are keenly aware of its existence and efficacy. One way to ensure that valid disputes are filed in practice despite their high costs is to set the required deposit during registration high enough to reimburse the issuer with (part of) the deposit of the offender, killing two birds with one stone.

Another aspect of ETHTID that can, to some extent, be addressed via economic means and the assumption of rational behavior on the side of council members is the recovery of the decryption key itself. Without any external pressure on the council as a whole, none of its members would want to release their share first, as withholding it confers a temporary position of advantage: When t shares have become public, any member holding an undisclosed share can privately recover the decryption key and thus access the contents of to-be-disclosed messages before anyone else. An ETHTID smart contract can minimize the window of opportunity to abuse such a position by incentivizing the council as a whole to recover and publish the decryption key quickly, for example by decreasing the reward for the council based on the number of blocks that pass between the scheduled and actual publication of the decryption key. In applications like the one we present in the next chapter, where coordinated disclosure is used as a means to document the independent creation of statements, the window for committing such statements should therefore close a few blocks before the decryption key is scheduled to be published.

Despite our goal to not rely on a centralized trusted party to realize a coordinated disclosure mechanism, ETHTID does include one role of particular importance that warrants a closer examination: the initiator. Before deploying an ETHTID contract instance, an initiator has the opportunity to not only set required protocol parameters like the threshold and phase schedule, but also to include disruptive functionality in the contract to, for example, steal the deposits of unwitting participants by self-destructing the contract prematurely. However, after deploying such a deceitful contract to the Ethereum blockchain, it is available for anyone to examine and uncover its malicious features. Overcoming the difficulties involved in such an examination falls within the research effort of extending software examination methods and tools to smart contracts [2].

Lastly, it is worth discussing a fundamental assumption underlying ETHTID, namely the independence of council members. The first-come-first-serve registration mechanism we employed in our proof of concept is plainly vulnerable to Sybil attacks [27]. In our paper [87, 86], we suggest the option of an initiator manually selecting council members for their instance, but this merely shifts the onus from the smart contract to the initiator and does not provide a sufficient solution to the Sybil problem. There are some approaches in the literature to achieve a certain Sybil resilience within decentralized systems, like the work by Gupta *et al.* [39, 38], but their suitability for applications like ETHTID remains an open question. In addition to such intrinsic approaches, the possibility of extrinsic solutions to the Sybil problem is also noteworthy. In particular, identities on distributed ledgers could be backed by government institutions to prevent natural persons from controlling more than one verified identity. One such approach was recently described by Maram *et al.* [55] which tasks a permissioned committee with processing credential requests. Through cryptographic means like secret sharing and secure multi-party computations, such requests contain government-issued unique identifiers like social security numbers or tax identification numbers that are then used for deduplication. Based on the assumption that the included identifiers are unique, not easily forgeable, and securely provisioned, attempts by natural persons to create multiple independent Sybil identities are stifled. The previously mentioned cryptographic techniques also ensure that the committee does not learn sensitive or personal data about credential requesters. Like in ETHTID, this last point also rests on the assumption that a sufficiently large portion of the committee is honest, incorruptible, and immune to compromise.

4.7.1. Limitations & Future Work

Similar to Palinodia, our focus with ETHTID was to demonstrate feasibility first and leave further optimizations as future work. Prime among such improvements is to make ETHTID instances capable of running more than one coordinated disclosure protocol, either sequentially or even in parallel. By reusing existing instances, deployment costs can be amortized over multiple executions and certain operations like council member registrations may not have to be repeated each time. Supporting sequential executions is markedly simpler as the registered council can change neatly between executions due to resignations, registrations, or disqualifications and there are no side-effects between executions. Parallel execution, meanwhile, presents a myriad of possible scenarios that have to be handled properly by the contract implementation. For example, a council member participating in two executions can be disqualified in one via a dispute, which could reveal a valid shadow in the other execution if the keys submitted during registration are reused and the disputing member also participates in both executions. Managing the status of registered members also becomes more complex compared to the sequential case with members joining before participating in any execution, being locked in while participating in at least one execution, and signaling their resignation before completing executions they are currently involved in. Having only few reusable ETHTID instances also limits the amount of validation effort and curtails the abilities of initiators of protocol executions to deceive participants: Since they can only set required parameters and not alter the general logic of the already deployed contract instance, scenarios as the one sketched in the discussion above are ruled out.

We noted on several occasions the integration of the incentive scheme by Yakira *et al.* [91] as future work. They examine scheduled disclosure scenarios similar to our problem statement via an "Escrow-DKG" protocol derived from the work of Pedersen [72] and Feldman [30] that includes more opportunities for participants to file disputes against each other compared to ETHTID. However, their focus is less on a practical implementation and more on the game-theoretic modelling of incentives and punishments to direct the behavior of participants and to discourage collusions. One noteworthy contribution of Yakira *et al.* concerns the collusion dynamic between participants. They show how collusions can be discouraged through *framing*, i.e. one colluding party publishing evidence of an active collusion to punish all participants and gain a small reward. In Subsection 4.6.2, we explain the crux of threshold cryptosystems with regards to assigning definitive blame for unintended behavior in more detail than Yakira *et al.*, but the end result is the same: In case of a proven collusion, all participants must be economically punished. Due to the differences between Escrow-DKG that they base their game theoretic analysis on and ETHTID, integration of both concepts is not entirely straightforward, especially in conjunction with making the contract reusable as described above, which turns the protocol from a one-time game into a repeatedly played game that changes the game-theoretic modelling significantly. Many of the complaints Yakira *et al.* use to economically punish offenders and reward reporters are either obviated through ETHTID's construction or they are issued and handled implicitly during a given protocol execution. For example, in their Escrow-DKG, Yakira *et al.* separate the submission of a hash of a participants contribution e_i and the submission of the contribution e_i itself, which necessitates complaints if the second transaction does not happen or the hash does not match. In ETHTID, a hash of e_i is never submitted and council members that registered but did not distribute their shadows in time can be handled during `generate_e` without separate complaints.

Lastly, future work on ETHTID may also include taking advantage of improvements to Ethereum, particularly the adoption of EIP-1829[16], EIP-1962[17], or a similar EIP, that would allow ETHTID to use a better-fitting and potentially less costly elliptic curve in its construction. In this way, we present an additional use case to motivate further development of Ethereum.

[16] `https://eips.ethereum.org/EIPS/eip-1829`
[17] `https://eips.ethereum.org/EIPS/eip-1962`

5. ETHDPR: Decentralized Public Review and Attestation of Software Attribute Claims on Ethereum

The content presented in this chapter has been published previously in the open access journal IEEE Access under the title “Decentralized Review and Attestation of Software Attribute Claims” by Stengele, Westermeyer, and Hartenstein [84].

With Palinodia and ETHTID thoroughly presented in the previous two chapters, we now face the task of integrating these components into a system for decentralized public reviews of software releases that we outlined in Chapter 1. The goal of such a system is to facilitate the review of binaries from initialization and execution to the recording of results without introducing a trusted third party. Ideally, this integration should not sacrifice any properties of its constituent components and be cost efficient. With Palinodia and ETHTID being built on Ethereum, we opted to design and implement our decentralized public review system ETHDPR on Ethereum as well.

5.1. Problem Statement

It is helpful to first expand on the rough problem statement from Chapter 1. Given a particular binary of a software, the task at hand is to enable a self-selected group of reviewers to examine said release regarding a specified set of attribute claims and independently record their assessment on a distributed ledger. Having potential reviewers decide for themselves whether or not to participate in the review of a given software release removes the need for a party with the power to grant and rescind this privilege but adds complexity to the processing of results, as we discuss later in this chapter. For the purpose of our work, we broadly define software attributes as statements about a binary that can be verified, falsified, or judged by a human actor either innately or through the use of tools. In this way, software attributes can concern the functionality, performance, or security properties of binaries. The main reason for performing such reviews is to enable relying parties, e.g. users of a software, to have greater confidence in a given binary without needing to perform the corresponding examination themselves. In the case of end users, it is worth noting that this practice lessens the trust in software developers described in Chapter 3 to a certain extent, following the adage “trust, but verify”.

In order for the results of a software review to be meaningful and useful, we specify a set of five objectives that we strive to fulfill with our concept and implementation [84]:

O1 Reviews should be created independently.

O2 Review process should be censorship resilient.

O3 Review process should be transparent[1] and enable traceability of artifacts.

O4 Review artifacts, including results, should be identifiable, persistently available, and traceable to the review instance and the software release under review.

O5 Possible attribute claims and review consolidation methods should be use case agnostic.

Note that we use the term *review* to refer to both the process and its result. With *artifacts*, we describe all digital objects necessary for or generated as part of a review process, including software binaries and their source code, specifications, and review results.

The first three objectives deal with the review process as a whole. To avoid single points of trust and to put reviewers under mutual competition to perform reviews thoroughly, each review should be performed by more than one reviewer. This in turn necessitates the independent creation of results as described in **O1**. By *independent*, we mean that each reviewer must finalize their results without knowledge of any other results pertaining to the same review. Next, in order to prevent reviews from being maliciously skewed, the process as a whole should be resilient against censorship (**O2**). In particular, this means that both the announcement of a review as well as the entirety of elicited results should ultimately be available to anyone. Lastly, **O3** encapsulates the need for transparency and the ability to relate review artifacts to each other. With *transparency*, we describe the ability of any party to observe all steps of a review process. *Traceability*, meanwhile, encompasses the ability of any party to understand the provenance and authorship of and relations between artifacts pertaining to any particular review process.

With **O4**, we group together objectives regarding artifacts themselves. Similar to the subject of a review, i.e. a binary, and all involved roles, artifacts necessary for or created during a review must also be uniquely identifiable in order to be traceable, for example. Next, the persistent availability of artifacts is of vital importance. A review cannot be performed if some of its necessary artifacts are unavailable and, similarly, a review whose results are (partially) unavailable afterwards is of diminished value. With identifiability and availability covered, the last overarching objective for artifacts is to take advantage of the aforementioned traceability and unequivocally connect artifacts to the software under review as well as the respective review instance.

Lastly, **O5** describes our aim to construct a system that can accommodate a wide spectrum of attribute claims. Put differently, the way a review process is executed on a distributed ledger should not restrict the kinds of attribute claims that can be examined and attested.

[1] Transparent in the sense of enabling any party to observe and scrutinize all aspects of review processes.

The remaining chapter is structured as follows. In Section 5.2, we review related work. With Section 5.3, we explain our system model, including roles of active participants, and the underlying attacker and trust model. We describe our concept and implementation of a decentralized public review system (ETHDPR) for software binaries based on Palinodia and ETHTID in Section 5.4 before evaluating it in Section 5.5. In Section 5.6, we provide a generalization of ETHDPR to give a concise description of its modular construction, thereby facilitating adaptation and future improvements of the whole system by improving or replacing individual components. We end the chapter with a discussion, an overview of the limitations of our work, and remaining open questions in Section 5.7.

5.2. Related Work

The practice of reviewing software for the purpose of quality control is well-established in the literature [76, 75]. More closely related to the approach presented here is the practice of software certification [46, 23, 41] to attest, for example, compliance with standards or regulatory requirements. While these works describe *what* a software review or certification entails, the work we present here investigates *how* such reviews can be performed without a centralized trusted party by utilizing distributed ledgers.

Another approach to ensure the safe use of software was proposed by Necula *et al.* in the form of proof-carrying code (PCC) [63, 64]. As the name implies, the core idea is for a software creator to attach proofs to code that can be checked efficiently on a user's system before installing or running it. Such proofs can, for example, show that a given piece of software will not consume more than a certain amount of memory during execution or that it will never access memory outside a specified address space. In this way, the adherence of new software to policies regarding resource or data usage set by a user's system can be ensured. The approach we present here is both contrasted and complementary to proof-carrying code. Proof-carrying code places the "burden of proof" on the software creators before the release of a binary whereas our approach is to enable the review of a published binary by parties other than its creator. The statements about software that PCC can support are limited but their validity can be checked with certainty and fully automatically on a user's system. Our approach places no restrictions on the types of software attribute claims that can be evaluated by requiring the active participation of multiple human reviewers. As a consequence, the approach in this work does not aim to achieve certainty on the attested software attribute claims but instead enables relying parties to judge the reliability of individual results in comparison to the results of other reviewers.

At a glance, initiating, conducting, and recording a software review on a public ledger bears similarities to blockchain oracles [1, 42] in that answers to specific questions should be determined and recorded on-chain. The crucial difference between blockchain oracles and decentralized software reviews as we conduct them in this work lies in how their results are being used: Oracles feed real-world information like weather data, results of sport competitions, or stock prices to smart contracts such that on-chain transactions that depend on this data can be executed and funds moved accordingly. Meanwhile, the results of software reviews, while being recorded on-chain, are primarily meant to be aggregated and used by relying parties, i.e. users of a software, without the need for any on-chain enforcement. As we describe later in this chapter, in our implementation, the results of reviews are in fact entirely inaccessible to smart contracts. The advantage we gain in turn are low and constant costs for logging review results on-chain, thereby allowing for arbitrarily large or complex attributes to be elicited. This difference in use of results directly influences the requirements placed on their creation. Since the distribution of funds can depend on the veracity and accuracy of data provided by oracles, great care must be taken to harden them against compromise or misuse. By contrast, individual reviews of a software release do not immediately affect any critical on-chain construction and relying parties can decide for themselves how much credibility they assign to each one.

Lastly, we re-review two works from Chapter 3 and show how their functionality beyond establishing identities for software relate to ETHDPR: Chainiac by Nikitin *et al.* [66] and SmartWitness by Guarnizo *et al.* [37]. While Chainiac does not include the evaluation of to-be-released binaries regarding arbitrary attributes, great attention is given to the "source to binary correspondence" via collectively trusted build servers and reproducible builds. In the approach presented here, the same correspondence between code and binary can be checked via reproducible builds as part of a review in addition to other attributes. SmartWitness distinguished itself from Palinodia by allowing accredited security providers to rate the security of binaries registered with a given SmartWitness instance. Supposedly due to logging said ratings on-chain, Guarnizo *et al.* limited these ratings to numerical values between zero and ten and, unlike ETHDPR, there is no mechanism coordinating the release of assessments by multiple providers. Consequently, later ratings in SmartWitness may be influenced by ratings submitted earlier.

5.3. System Model

As with the previous two chapters, we begin by giving an overview of our system model, particularly the involved roles depicted in Figure 5.1 and our assumptions regarding attackers and trust. Similar to ETHTID, our focus in this work is to demonstrate the feasibility of conducting decentralized public reviews of software releases on distributed ledgers and, as such, certain aspects like incentivization or reputation management of reviewers are left for future work.

5.3.1. Roles

The role of *maintainer* is inherited from Palinodia and was described in Section 3.3. By creating and publishing new binaries of a software, maintainers contribute the primary object of a review. As a passing remark, maintainers can also incorporate past review results into the future development of their software. *Claimants* provide the secondary object of review in the form of attribute claims a given binary should be examined against.

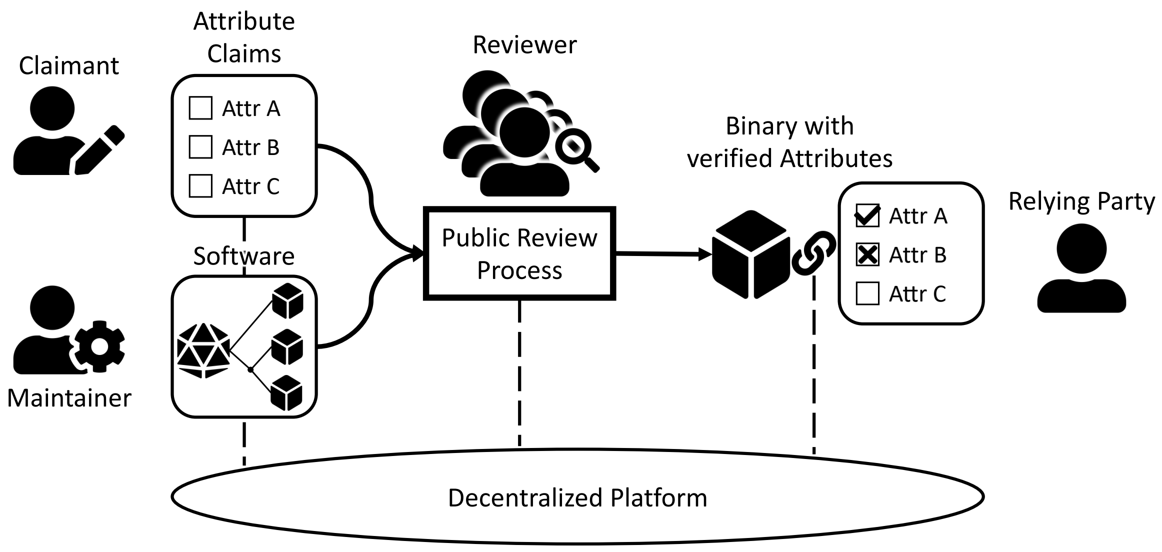


Figure 5.1.: Overview of ETHDPR. Maintainer publishes new binary and attaches it to their software identity. Claimant submits set of attribute claims to be reviewed. Reviewers examine the new release against the stated claims and publish their results. Relying parties can obtain binary with attached review results. [84]

For the sake of presentation, we assume that claimants state reasonable and suitable claims about software. Unrealistic claims should only pose a mild inconvenience in our approach and not threaten any of the objectives stated above. A self-selected set of *reviewers* examines binaries independently and publishes their attributions according to their individual findings. It is important to note the difference between a software attribute claim and an actual software attribute in the context of this work: An *attribute claim* is merely a way to elicit and direct the investigations as part of a software review. Without the corresponding results, a claim has no basis or merit on its own. A *software attribute* is a concise and subjective statement by a particular reviewer whether a certain claim about a given software binary holds or not. Lastly, *relying parties* make use of the resulting software attributes. In the case of end users, a collection of review results may inform their decision on whether or not to use a certain release of a software. Similarly, software maintainers can also be relying parties when it comes to using and building upon other software based on their attributes. As in previous chapters, none of the above roles are mutually exclusive and we term their union as *stakeholders* of a given software.

5.3.2. Attacker and Trust Model

We first describe the capabilities and limitations of our attacker model before describing possible goals.

Similar to the attacker models described in Subsection 3.3.2 and Subsection 4.4.2, an attacker for ETHDPR is computationally bound and cannot break cryptographic assumptions, in particular the first[2] and second[3] pre-image resistance properties of cryptographic hash functions and the computation of discrete logarithms. An attacker is also unable to prevent read and write access to the underlying blockchain for a period of time long enough for any other party to miss a particular deadline. For example, an attacker cannot prevent a reviewer from recording their results on-chain long enough for the review period to close. In the case of read accesses, there are no hard deadlines that can be missed. Instead, we assume that every attempt to read blockchain data must succeed eventually. Likewise, an attacker cannot gain significant control over the consensus mechanism of the underlying distributed ledger. In particular, an attacker cannot induce a fork in the ledger to change or delete past records and they cannot forcibly add invalid transactions to the ledger. Lastly, an attacker can interact with the application layer of said ledger just like any other user by deploying new smart contract instances or by interacting with existing instances.

The first possible goal of an attacker is to sabotage a review by preventing either the announcement or certain elicited results from reaching a particular user. In conjunction with Palinodia, such an attack could serve the purpose of having a user install and run software with vulnerabilities that were discovered during said review, thereby opening an avenue for the attacker to further compromise said user's system. Essentially, this attack serves to challenge the censorship-resilience objective **O2**. A second way for an attacker to undermine a review instance is to submit mutually contradictory results to different sets of parties with the goal of splitting their views of the results. In other contexts, especially consensus systems, such behavior is known as "equivocation".

Similar to ETHTID, the decentralized public review mechanism we propose here is immediately vulnerable to Sybil attacks [27] by reviewers looking to amplify their voice and maliciously skew results. We consider Sybil resistance a strongly related and crucial but ultimately out-of-scope problem for the present work.

5.4. ETHDPR

We now present ETHDPR, a decentralized public review system for software releases on Ethereum. With Figure 5.2, we provide an overview of an ETHDPR execution from the initialization of a review to its conclusion. We first describe an ETHDPR execution abstractly before illustrating it with an example in Subsection 5.4.1.

Before a review can be performed, a Palinodia software identity must be established on Ethereum by a developer and maintainer, particularly a Binary Hash Storage (BHS) contract as described in Subsection 3.4.1.2. Please recall that a HashID together with the address of the BHS contract instance where it was registered serve as both a unique identifier for a given software release as well as a way to obtain additional information about the corresponding software identity via the Ethereum peer-to-peer network. While

[2] Given a hash h, it is difficult to find a message m such that $h = \mathrm{H}(m)$

[3] Given a message m_1, it is difficult to find a message m_2 such that $\mathrm{H}(m_1) = \mathrm{H}(m_2)$

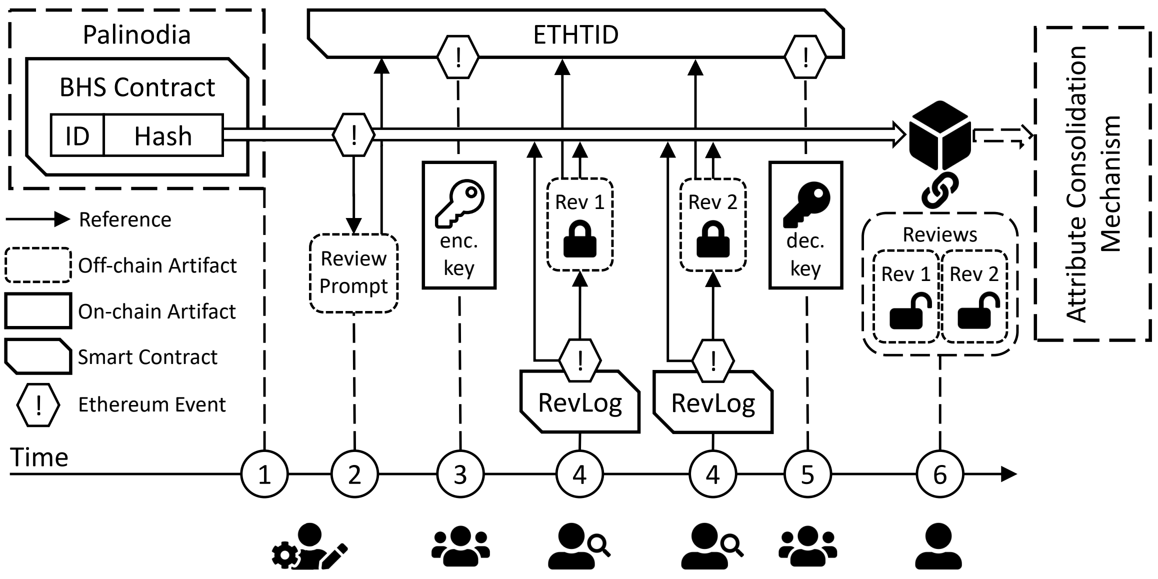


Figure 5.2.: Overview of actors, smart contracts, events, and artifacts in an ETHDPR execution. **(1)** Maintainer establishes software identity via Palinodia. **(2)** Maintainer publishes a new binary hash to a BHS contract referencing a review prompt containing attribute claims to be verified and a reference to an ETHTID instance for scheduling and disclosure coordination. **(3)** Council of ETHTID instance releases an encryption key. **(4)** Reviewers examine software release, encrypt their results via the ETHTID encryption key, and make to-be-disclosed results available via IPFS. Each reviewer time-stamps a reference to their review results within the review period through their own Review Log (RevLog) smart contract and link it to both the software under review and ETHTID instance for coordinated disclosure. **(5)** Council of ETHTID instance releases decryption key, thereby ending the review period. **(6)** Relying parties can obtain and decrypt review results associated with a software release. Additionally, attributes can be consolidated further by relying parties or other stakeholders. [84]

we introduced the roles of maintainer and claimant separately in Subsection 5.3.1, they are both occupied by the software maintainer responsible for a given BHS contract instance. In preparation for the release and review of their new software binary, a maintainer prepares a JSON-formatted *review prompt* to state both the claimed attributes of a new software release as well as the address of an ETHTID contract instance to enforce the review schedule. This can either be a preexisting instance with a suitable schedule or a newly created instance with a schedule chosen by the maintainer. Depending on the stated claims, CID based references to other digital objects, like the binary's source code or (in)formal specifications can also be part of the review prompt.

With a software identity established, a review for a new binary is initiated by the respective maintainer as part of the `publishHash` transaction, which stores the hash of the newly released binary as part of a BHS contract instance's state, keyed by a HashID. Rather than storing the CID of a review prompt also as part of the contract state, a significantly cheaper but equally suitable option is to include the CID within the `Publication` event that is emitted during the `publishHash` transaction. This construction is sensible as CIDs only need to be logged on-chain to be retrieved by clients and they never serve as the basis for a

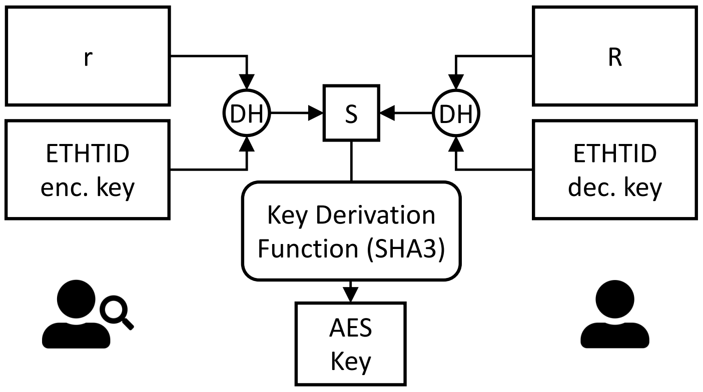


Figure 5.3.: A key derivation procedure akin to the Elliptic Curve Integrated Encryption Scheme (ECIES) [56]. To prepare their review results for a coordinated disclosure, each reviewer derives a unique AES key by first generating a random, ephemeral discrete log key pair (r, R) and then performing a Diffie-Hellman(-Merkle) key exchange [26, 59] with r and an ETHTID encryption key to generate a shared secret S. Through a key derivation function (SHA-3 in our case), each reviewer generates a unique AES key from S to encrypt their results. Each reviewer attaches R to their review, thereby enabling any relying party to recover S and the AES key once the corresponding ETHTID decryption key is available.

future change to a contract instance's state[4]. Note that this inclusion extends the purpose of the `Publication` event from merely announcing the release of a new binary to also serve as the cornerstone for a review. Consequently, the event needs to be easily retrievable both during and after each review. The way we chose to ensure this retrievability is to add the HashID to the indexed parameters of the `Publication` event. Since every Palinodia-registered binary includes metadata containing both its HashID as well as the BHS contract instance address where it is registered, a Palinodia client can issue a very targeted search query to its associated Ethereum client to retrieve a corresponding `Publication` event and see if a review is or was performed on this particular release.

Once the review prompt is announced on-chain and available via IPFS along with the software binary in question and other review artifacts, self-selected reviewers can begin their examination. It is important to note that, in accordance with our objective **O5**, the only restriction we place on the tools and methods used for a software evaluation is that both examination goals and results must be encodable as text, even if only as IPFS CIDs to images or other multimedia files. Reviewers compile their results in a JSON-formatted *review*, consisting of a header with necessary information and an encrypted payload containing the actual review results. More specifically, the header of a review contains both the BHS contract instance address and HashID to uniquely identify the reviewed binary as well as the ETHTID contract instance address where the decryption key to access the payload can be retrieved, once available. The last necessary component of the header concerns the encryption of the attached payload such that it can be decrypted once the ETHTID decryption key is released.

As we briefly mentioned in the previous chapter, a symmetric encryption key can be derived from an ETHTID encryption key via a Diffie-Hellman(-Merkle) key exchange, a common

[4] Event data is inaccessible to smart contract instances, even those that emitted said events.

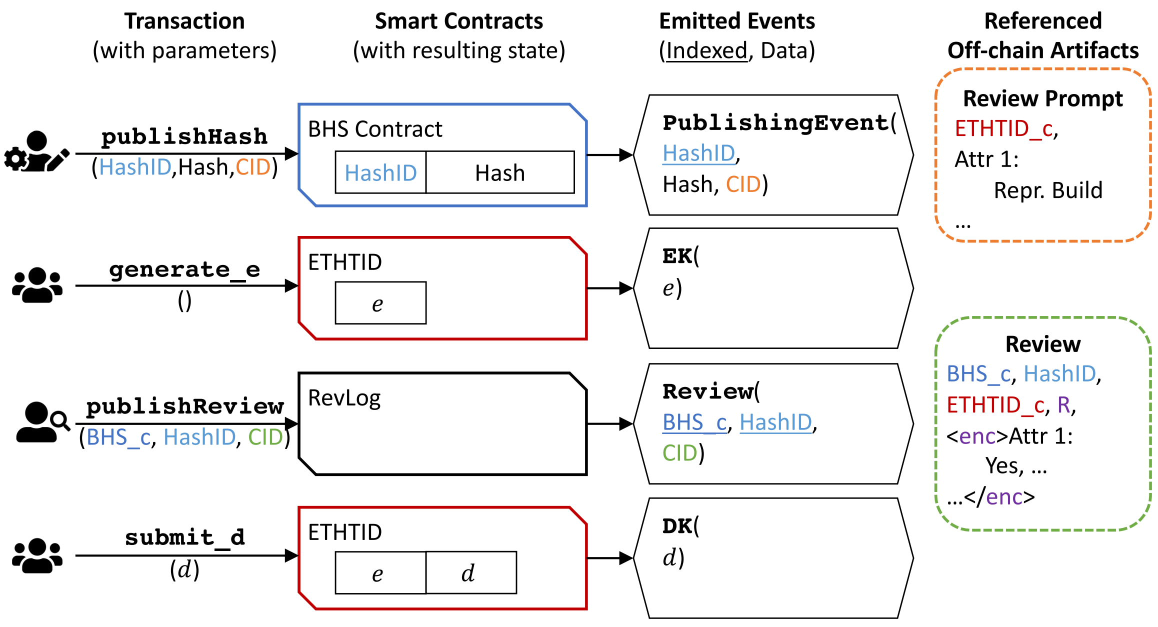


Figure 5.4.: Relations between transactions, smart contracts with their resulting states, emitted events, and referenced off-chain artifacts. Colors denote references, either as contract instance addresses (ETHTID_c, BHS_c), strings (HashID), or IPFS CIDs. Underlined event attributes are indexed for the purpose of monitoring and searching. On-chain artifacts are stored as part their respective contract instance's state and depicted as rectangular boxes. The purple value R in **Review** corresponds to the elliptic curve point of the same name from Figure 5.3.

approach to hybrid encryption called Elliptic Curve Integrated Encryption Scheme (ECIES), a simplified overview of which we provide in Figure 5.3. Basically, each reviewer randomly generates an ephemeral discrete log key pair $\langle r, R := g^r \rangle$, such that a key exchange between either r and the ETHTID encryption key or R and the corresponding decryption key lead to the same shared elliptic curve point S. By converting S into a normalized form[5] and canonical encoding, a 256 bit symmetric encryption key can be derived through a key derivation function like SHA-3. Each reviewer includes R in the header of their review and encrypts the payload with the derived key using symmetric encryption like the Advanced Encryption Standard (AES). The payload of each review consists of a reviewer's assessment of claims stated in the review prompt, primarily whether or not they attest to the binary in question fulfilling the stated claims. Further information can also be included for each claim, like additional assumptions that the reviewer had to make or which tools and settings they used in their examination. Note that the above key derivation can only be performed by anyone besides the author of a review once the ETHTID instance publishes its decryption key.

Similar to the review prompt above, individual reviews also need to be logged on-chain to prove their creation before the publication of the ETHTID decryption key. For this purpose,

[5] Points on elliptic curves can have more than one representation that are more efficient for certain operations, but they can always be converted to a unique normalized form. For a more detailed explanation, we refer to section 3.2 of "Guide to Elliptic Curve Cryptography" by Hankerson, Menezes, and Vanstone [40].

Listing 5.1: C source code of our toy example software. The code implements a precision-loss calculation for the state of charge in an electric vehicle. It computes the rounded floor value `soc_rounded` of a decimal number encoded as the integer `soc` via `factor`.

```
int encoded_floor(int soc, char factor) {
  if (soc < 0) {
    return -1;
  } else if (factor == 0) {
    return -1;
  }
  int max = 101 * factor;
  if (soc >= max) {
    return -1;
  }

  int soc_point =  soc % factor;
  int soc_rounded =  soc - soc_point;

  return soc_rounded;
}
```

we employ a minimalist Review Log (RevLog) smart contract for each reviewer. Each RevLog contract instance only provides one access-controlled function to emit a `Review` event containing a BHS contract instance address and HashID as indexed parameters and an IPFS CID of the review as data. The emission of such events ties together the Ethereum address of a review author with an integrity-protecting reference to their results alongside the aforementioned time-stamping. This way, anyone in possession of a Palinodia-registered binary can efficiently recover not only the CID of a review prompt, if one was published, but also the CIDs of all submitted reviews by instructing their Ethereum client to search for corresponding events in the block-range specified by the referenced ETHTID instance. With these CIDs, both review prompt and elicited reviews can then be obtained and their integrity verified via an IPFS client. With Figure 5.4, we give an overview of how review prompt, reviews, and contract instances are interconnected.

With the publication of the ETHTID decryption key, the submission period for reviews ends and the payload of properly prepared reviews can be decrypted. Depending on the needs of relying parties and the nature of reviewed and attested claims, the results of reviews can be consolidated in different ways. As we focused more on the execution and coordination of software reviews, we can only give suggestions on possible consolidation methods, in part also because some form of reputation management or credibility score for reviewers is necessary as well. For binary claims, a majority vote may be suitable, whereas for claims on a gradient, a weighted average could be fitting. For certain claims, the existence of a single valid counterexample may overrule any other consolidation strategy. We elaborate this point further in Section 5.6.

Listing 5.2: Example review prompt. A review prompt uniquely references a review instance by stating its corresponding ETHTID instance address as well as the BHS contract instance address and HashID of the software binary under review. The purpose of a review prompt is to state the claims that should be investigated in this review instance. Claims may reference additional information in the form of claim artifacts. In the example we abbreviate the values of references with ellipses for the sake of readability.

```
{
  "ethtid_contract_addr" : "0x71d...",
  "bhs_contract_addr" : "0xb85...",
  "hashID" : "PrecLoss-1.0",
  "claim_artifacts" : {
    "source_code" : "QmWj2...",
    "build_info" : "QmRju...",
    "formal_specification" : "QmaZu...",
    "frama-c_info" : "QmTCj...",
    "informal_specification" : "QmUBY..."
  },
  "claims" : {
    "c1" : "Build is reproducible from ${source_code} using ${build_info
       }",
    "c2" : "Formal verification of the ${source_code}'s precision-loss
       functionality is reproducible in Frama-C with ${
       formal_specification} and ${frama-c_info}.",
    "c3" : "${formal_specification} of precision-loss functionality is
       valid with respect to the software's ${informal_specification
       }.",
    "c4" : "Potentially unwanted unspecified behavior was not detected
       in ${source_code} with respect to the software's ${
       informal_specification}."
  }
}
```

5.4.1. Illustrative Example

To better illustrate the construction and interactions between parties during an ETHDPR execution, we provide the following example. Sharing precise information about a device's charge level have lead to privacy issues in the past [69, 68]. The same has recently been shown for internet-of-things devices [51] and electric vehicles [16]. Our toy software example, the source code of which is depicted in Listing 5.1, implements a function to reduce the precision of the state of charge of an electric vehicle so that this information can be shared with services outside the vehicle while better protecting the car owner's privacy. In addition to using Palinodia to enable car owners to verify the authenticity of software running in their vehicles, an original equipment manufacturer (OEM) now also uses ETHDPR to demonstrate to both end users and other relying parties that the software does exactly what it is supposed to do and nothing else.

In addition to the source code and build info, both a formal and an informal specification of the software's functionality are made available via IPFS for review purposes. The OEM maintaining this software states the following claims for the purpose of this example in their review prompt depicted in Listing 5.2:

Listing 5.3: Formal Specification in the ANSI/ISO C Specification Language (ACSL) [8] of the exemplary review prompt claim artifact `formal_specification`. The OEM formally specifies the expected functionality of the software binary under review and corresponding `source_code` (see Listing 5.1).

```
/*@ assigns \nothing;
  behavior invalid:
    assumes soc < 0 || factor < 1
      || soc >= 101 * factor;
    ensures \result == -1;
  behavior valid:
    assumes soc >= 0 && factor >= 1
      && soc < 101 * factor;
    //valid result range
    ensures 0 <= \result <= 100 * factor;
    //soc_rounded is rounded
    ensures \result % factor == 0;
    //soc_rounded is floor of soc
    ensures \result <= soc
      < \result + factor;
  complete behaviors invalid, valid;
  disjoint behaviors invalid, valid;
*/

int encoded_floor(int soc, char factor);
```

Listing 5.4: Frama-C/WP plug-in command of the exemplary review prompt claim artifact `frama-c_info` that specifies how the OEM executed the claimed reproducible formal verification of the code in Listing 5.1.

```
frama-c -wp -wp-rte example.c \
  -then -report
```

C1 The distributed binary can be built reproducibly[6] from its source code

C2 The implementation of the precision-loss functionality can be verified formally

C3 The formal and informal specifications are congruent and sound

C4 There is no hidden or unspecified functionality present

For the purpose of this example, we focus on claims **C2** and **C3**. Using the ANSI/ISO C Specification Language (ACSL) [8], the OEM formally specifies the intended functionality of the precision-loss function as shown in Listing 5.3, particularly regarding invalid and valid ranges of inputs and the correct rounding down of the output. To execute the formal verification of the C source code, the OEM recommends Frama-C with its weakest predicate (WP) plug-in [9] and provides the corresponding instruction in Listing 5.4 as the `frama-c_info` artifact in the review prompt. Note that other tools for this kind of verification could be used as well.

[6] From `https://reproducible-builds.org/`: “A build is reproducible if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts.”

Listing 5.5: Frama-C/WP plug-in verification report as an exemplary `attachment` explanation of a review result such as the attestation in Listing 5.6. Some details are omitted for readability.

```
[kernel] Parsing example.c
  (with preprocessing)
[rte] annotating function encoded_floor
[wp] 15 goals scheduled [...]
[wp] Proved goals:   15 / 15
 Qed:               5   (0.68ms-7ms-60ms)
 Alt-Ergo 2.4.1:   10   (1ms-26ms-191ms)
 [...]
----------------------------------------
--- Status Report Summary
----------------------------------------
    15 Completely validated
    15 Total
----------------------------------------
```

Listing 5.6: Example review result structure stating an attestation of claims **C2** and **C3** while abstaining from other claims of this review instance.

```
{
  "c1" : {},
  "c2" : {
    "attestation" : true,
    "attachment" "Qmb1d..."
  },
  "c3" : {
    "attestation" : true
  },
  "c4" : {}
}
```

Listing 5.7: Example review structure. Analogous to a review prompt (see Listing 5.2), a review is uniquely referenced by the CID of its corresponding review prompt, ETHTID contract instance address, and software binary. The review also contains the encrypted results (see Listing 5.6) as well as an elliptic curve point to recover the AES key with which the results can be decrypted once the ETHTID decryption key becomes available.

```
{
  "review_prompt_cid" : "QmYGc...",
  "ethtid_contract_addr" : "0x71d...",
  "bhs_contract_addr" : "0xb85...",
  "hashID" : "PrecLoss-1.0",
  "ethtid_ecies_aes_key" : "(1298..., 78152...)",
  "encrypted_results" : "..."
}
```

After the new precision loss binary is released and the review is announced with the CID of the review prompt shown in Listing 5.2, a reviewer with experience in Frama-C decides to participate in the review and attestation of claims **C2** and **C3**. They examine the specification in Listing 5.3 and compare it to the informal specification provided under the `informal_specification` reference in the review prompt to decide on their attestation of claim **C3** and, for the sake of this example, find the specifications to be sound and congruent. They then execute the formal verification as specified in Listing 5.4 against the source code from Listing 5.1 to decide on claim **C2**. They find that the verification passes successfully, the abbreviated output of which is shown in Listing 5.5. Based on these results, they compile their review payload as shown in Listing 5.6, where they also include a reference to their Frama-C output as an attachment to their attestation of **C2**. Note that reviewers can abstain from attesting claims, **C1** and **C4** in this example. Finally, the reviewer derives a unique AES key as depicted in Figure 5.3 and uses it to encrypt their results from Listing 5.6 to construct their review as depicted in Listing 5.7. They then log, and thereby announce, their review through their RevLog contract and make it available via IPFS for other relying parties to obtain in preparation for its scheduled disclosure.

Once the review period ends with the release of the ETHTID decryption key, relying parties can decrypt and consolidate the results of all submitted reviews based on their needs. Claim **C1** is of particular note in this regard, as a verified source-to-binary correspondence through reproducible builds allows the transference of claims between source code, i.e. claims C2-4 in this example, and the distributed binary. However, as shown in the above example, not all reviewers must attest to all claims in their reviews. Instead, it would be sufficient for a later consolidation of attributes that each claim was examined and attested by some credible reviewers.

5.5. Evaluation

Analogous to the previous two chapters, we constructed ETHDPR based on Palinodia and ETHTID, together with the RevLog contract, in Solidity 0.8[7] and deployed it to a local development blockchain on the London hard fork using Ganache v7.2.0 of the Truffle Suite development tools. Similar to ETHTID, we also implemented the off-chain operations of reviewers and relying parties, particularly the generation, encryption, and decryption of reviews, as a Python application. We use IPFS CIDs to estimate realistic costs for the on-chain storage of references to files located off-chain and otherwise exclude the performance of IPFS from the evaluation of ETHDPR. For a detailed evaluation of the performance of IPFS, we refer to the work by Shen *et al.* [81].

As before, we first provide a quantitative evaluation of gas costs and relevant off-chain operations before examining ETHDPR's security properties.

[7] `https://git.scc.kit.edu/dsn-projects/dissertations/dsim/-/tree/main/ETHDPR`

Table 5.1.: Cost of publishing review prompt IPFS CIDs alongside binary hashes to BHS contract instances and publishing CIDs of reviews through RevLog contract instances. The length of HashIDs included in either transaction type is fixed at ten characters. The cost of `publishHash` with a CID length of zero corresponds to the Palinodia base case from Table 3.4. Conversion of gas to USD via daily average exchange rates for 1st June 2022 as reported by Etherscan: USD 1817.42 per ETH, ETH 60.06×10^{-9} per gas.

	`publishHash`		`publishReview`	
CID Length	Gas	USD	Gas	USD
0	89 667	9.79	-	-
46	92 135	10.06	29 471	3.22
60	92 301	10.07	29 639	3.24
66	92 831	10.13	30 171	3.29
111	93 835	10.24	31 171	3.40

5.5.1. Gas Costs & Performance

To examine the practical feasibility of our approach, we tackle the on- and off-chain components differently: As with previous chapters, we examine the gas costs for on-chain operations to judge their practicality. While the elliptic curve integrated encryption scheme we employ for the off-chain encryption and decryption of review payloads is common practice, it is not generally used with the Barreto-Naehrig curve that Ethereum currently limits us to. As such, we perform conventional performance tests to demonstrate that, even with an unconventional curve, the effort of encrypting and decrypting sets of review payloads remains acceptable.

In Table 5.1, we show the costs of publishing CIDs for review prompts and reviews. In Table 5.2, we present a cost overview of Palinodia and ETHTID and the increases to deployment and operational costs that arose through their integration into ETHDPR. Note the effect of the changes to the `Publication` event we described in Section 5.4: Changing the included HashID from an unindexed to an indexed event parameter and including a CID as a string adds a minimum of 2468 gas to the overall transaction costs. Also of note is a subsequent increase to the deployment costs of BHS contracts from 1 077 213 to 1 129 719, an increase of 52 506 gas or USD 5.73. Compared to adding a CID to events that would be emitted regardless, publishing CIDs to reviews is markedly more expensive, despite the corresponding transactions merely verifying the sender as the owner of a RevLog contract instance and emitting an event. For each RevLog contract instance, we observe deployment costs of 253 815 gas or USD 27.70. The disproportionately large increases in costs between CIDs of length 60 and 66, as well as between 66 and 111 in either scenario above are due to a padding of strings and other parameters to the nearest 32 B as defined by the Ethereum contract ABI specification[8]. As IPFS CIDs only use ASCII characters, the length of strings in characters and bytes is interchangeable. Table 5.2 succinctly shows that, compared to the deployment and operational costs of Palinodia and ETHTID, the increases

[8] `https://docs.soliditylang.org/en/develop/abi-spec.html`

Table 5.2.: Cost overview of integrating Palinodia and ETHTID into ETHDPR. A “-” under Integration denotes that no changes were necessary and thus costs remained unchanged. A “-” under Initial Version denotes that the respective contract or function was added during integration. All omitted functions of Palinodia contracts remain unchanged. Conversion of gas to USD via daily average exchange rates for 1st June 2022 as reported by Etherscan: USD 1817.42 per ETH, ETH 60.06×10^{-9} per gas.

Deployment Costs	Initial Version		Integration		Delta		
Contract	Gas	USD	Gas	USD	Gas	USD	Comment
SW	1 143 544	124.81	-	-	-	-	One per software
BHS	1 077 213	117.57	1 129 719	123.31	52 506	5.73	One per maintainer
IDM	927 201	101.20	-	-	-	-	One per developer/maintainer
ETHTID	1 879 437	205.13	-	-	-	-	One per review period
RevLog	-	-	253 815	27.70	253 815	27.70	One per reviewer
Operational Costs							
Function	Gas	USD	Gas	USD	Gas	USD	Comment
`publishHash`	89 667	9.79	92 301	10.07	2634	0.28	Once per binary. 10 char HashID. Integrated version with CID length of 60.
`publishReview`	-	-	29 639	3.24	29 639	3.24	Once per submitted review. CID length of 60.
`revokeHash`	38 849	4.24	-	-	-	-	10 char HashID.
ETHTID execution	4 559 309	497.63	-	-	-	-	$n = 16, t = 7$, happy case, excluding deployment.

Table 5.3.: Mean decryption time in seconds (rounded to two decimals) of different sized review sets with different review payload sizes over 100 repetitions. [84]

Review Set Size	Attachment Size per Review in MB					
	0	0.2	0.4	0.6	0.8	1.0
10	0.10	0.12	0.14	0.17	0.19	0.21
20	0.21	0.25	0.29	0.33	0.37	0.41
30	0.31	0.38	0.44	0.51	0.57	0.63
40	0.41	0.49	0.58	0.67	0.75	0.84
50	0.52	0.63	0.72	0.84	0.95	1.06

to both costs as a consequence of their integration into ETHDPR is rather small. It is particularly noteworthy that ETHTID remained entirely unchanged as it was conceived from the start to be a neatly encapsulated service to other decentralized applications. Comparing deployment to operational costs also reinforces the future goal of making contract instances as reusable as possible that we raised in Subsections 3.6.1 and 4.7.1.

To evaluate the effort for encrypting and decrypting reviews of varying size, we used an OpenStack virtual machine running Ubuntu 20.04 on four virtual 2.4 GHz AMD EPYC CPUs and 8 GB RAM with Python 3.8.10. Just like in the ETHTID Python application, we use the `py_ecc` module[9] for elliptic curve operations and combine it with the SHA-3 implementation of the `hashlib` Python library[10] to derive symmetric encryption keys from ETHTID encryption and decryption keys. In the following evaluation, we use the implementation of the Advanced Encryption Standard (AES) in Galois Counter Mode (GCM) of the `cryptography` Python module[11] for the encryption and decryption of review payloads.

We generate reviews with random payloads of varying sizes and average the time for their encryption and decryption over 100 iterations to curb anomalies and outliers. For the encryption of reviews, which includes the key derivation shown in Figure 5.3 and the AES encryption of their payload, we observe a linear correlation between payload size and run time: We measured a baseline of 0.02 s for reviews without a payload to encrypt, i.e. just the key derivation, and 0.18 s for a review with a 1 MB payload.

The decryption of review payloads is somewhat more intricate. A relying party compiling reviews for a given binary will most likely have to decrypt multiple payloads once the ETHTID decryption key becomes available. Additionally, cases of "blind plagiarism" can be identified during the decryption process. While reviewers cannot access the contents of other reviewer's submitted payloads, they can trivially copy both the attached curve point for key derivation and the encrypted payload in its entirety to blindly copy someone else's results without actually performing any work themselves. As simple as such a plagiarism

[9] `https://github.com/ethereum/py_ecc`
[10] `https://docs.python.org/3/library/hashlib.html`
[11] `https://cryptography.io/`

is to perform, so too is it simple to detect: While processing reviews in the order they were logged on-chain, the attached ECIES curve points are stored after checking their uniqueness compared to previously stored points. Reviews that contain an ECIES curve point from a previously processed review are then ignored. In Table 5.3, we provide the average time to process 10 to 50 reviews with payload sizes between 0 and 1 MB, including the aforementioned plagiarism check and payload decryption. We observe a linear growth in decryption time for both the number of processed reviews and payload size.

5.5.2. Functional & Security Considerations

We now compare our implementation of ETHDPR against the objectives we stated in the beginning of the chapter, which we repeat here for convenience.

O1 *(Reviews should be created independently)*: The independent generation of reviews is enabled and supported in ETHDPR via a coordinated information disclosure mechanism, but it cannot be enforced or guaranteed. Consequently, the fulfillment of this objective rests on the assumption that self-selected reviewers are willing to properly use the disclosure coordination mechanism. A malicious reviewer looking to cheat off other reviewer's results without their cooperation would have to compromise a sufficiently large portion of the associated ETHTID instance's council to obtain the necessary decryption key early. Lastly, detecting and punishing collusions via on-chain mechanisms is currently not an option in ETHDPR, primarily due to reviews being stored off-chain and thus inaccessible to smart contracts.

O2 *(Review process should be censorship resilient)*: ETHDPR's resistance against censorship must be examined separately for its on- and off-chain parts. Similar to replay and freeze attacks on Palinodia that we examined in Subsection 3.5.2, an attacker looking to hide the announcement of a software review or particular attributions would have to break fundamental properties of the underlying distributed ledger. Preventing a particular user's Ethereum client from obtaining any blockchain data would raise suspicion after mere minutes as new blocks are expected to be generated every 13 s. Creating a convincing alternate chain where specific transactions are removed would require computational resources that exceed our attacker model assumptions. However, unlike Palinodia, on-chain records are only half the battle in ETHDPR, as they mostly contain references to off-chain artifacts available via IPFS. IPFS itself does not guarantee the availability of these artifacts. Instead, their creators and other stakeholders of a given software must store and disseminate them upon request. In addition to hosting their own review, it would seem sensible for reviewers of a particular software release to also store and provide both the review prompt as well as reviews of other participants. Censoring any particular artifact becomes increasingly challenging the further said artifact has been propagated and replicated. In summary, the long-term availability of off-chain artifacts rests upon the assumption that the willingness of reviewers to participate extends to the hosting and sharing of artifacts, both their own and those created by other reviewers, in addition to the general functionality of IPFS.

Akin to censorship, an equivocation attack by a malicious reviewer aiming to split the public view on a particular release by submitting two or more mutually contradicting attestations faces similar hurdles. Without breaking the fundamental properties of the underlying ledger as explained above, everyone will see all attestations by such an attacker, thus foiling their attack. Alternatively, an attacker may attempt to provide contradictory attestations under a single CID, which would require a hash collision, thereby violating the second preimage resistance property of cryptographic hash functions. However, significantly easier than the above approaches, an equivocation attacker could instead create Sybil identities to submit contradictory attestations. Without a Sybil-resistant reviewer identity system, ETHDPR in its current form remains vulnerable to such attacks with two mitigating factors: First, the costs for such an attack scale linearly with the number of established Sybil identities and the number of logged attestations. And second, based on the fundamental properties of distributed ledgers, the entire attack must be executed in plan view of the public, where attentive observers may draw attention to it.

O3 *(Review process should be transparent and enable traceability of artifacts)*: Note how a distributed ledger with its fundamental properties, most prominently a highly available public consensus on one world state per block, provides a base layer of transparency for ETHDPR upon which the traceability of artifacts is built. Through transactions and events, critical steps like the announcement of a review, the beginning and end of the review period, and the time-stamping of reviews are unequivocally and irrevocably written to a tamper-proof public log. However, in order to take advantage of this transparency for the purpose of traceability, participants must adhere to a common protocol, i.e. the name and structure of events containing correct references to contract instances and off-chain artifacts. For example, a review that is logged on-chain in a protocol-deviating way will not be found by protocol-adhering parties. Generally, misbehavior from claimants/maintainers or reviewers in ETHDPR is mainly detrimental to the respective party in the sense that review announcements or reviews, while being logged on-chain, are not discoverable by parties following the protocol. However, much like the availability argument above, the ledger on its own is not sufficient in the case of ETHDPR due to artifacts being stored off-chain. Assuming artifacts are available via IPFS, their traceability is again dependent on their creators including the specified references to other artifacts. In summary, transparency in ETHDPR relies on both the properties of public ledgers and participants' willingness to use it properly. Traceability, meanwhile, requires transparency and additionally relies on participants to properly include the necessary references and to provide and disseminate artifacts via IPFS.

O4 *(Review artifacts, including results, should be identifiable, persistently available, and traceable to the review instance and the software release under review)*: ETHDPR derives these properties from already introduced assumptions. Review artifacts stored off-chain inherit unique identifiers from IPFS in the form of hash-based CIDs. To break this property, an attacker would have to generate two distinct files with the same CID, which is equivalent in difficulty to finding a collision in a cryptographic hash function. The availability of artifacts depends on the willingness of their creators and other interested parties to provide and disseminate them via IPFS, as explained above for objective **O2**. An attacker aiming to make the file behind a given CID unavailable would have to compromise all IPFS nodes

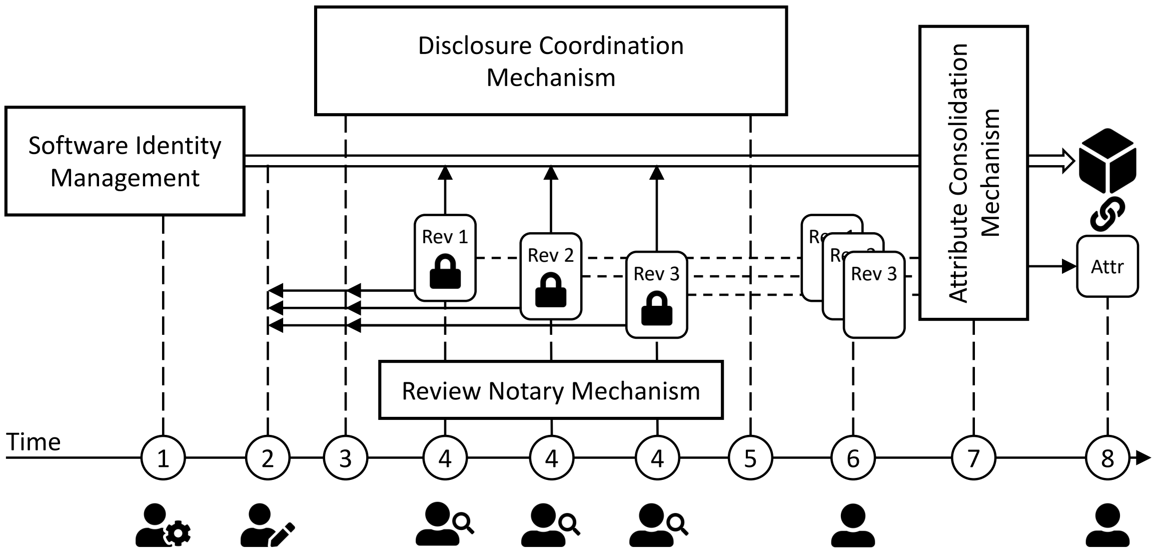

Disclosure Coordination Mechanism
Software Identity Management
Attribute Consolidation Mechanism
Rev 1
Rev 2
Rev 3
Attr
Review Notary Mechanism
Time
1
2
3
4
4
4
5
6
7
8

Figure 5.5.: A generalization of ETHDPR. **(1)** Maintainer establishes software identity. **(2)** Claimant initiates review by publishing attribute claims to be verified. **(3)** Setup for disclosure coordination mechanism completed, starting the review period. **(4)** Reviewers submit reviews through notary mechanism to timestamp them within the review period. **(5)** Disclosure coordination mechanism ends the review period, making contents of all appropriately prepared reviews public. **(6)** Relying parties can obtain review results and perform aggregation. **(7)** Review results are consolidated into concise attributes and attached to software identity. **(8)** Relying parties obtain consolidated attributes of a software binary. [84]

holding and providing said file or convince their owners to "unpin" this file and no longer provide it on request. Lastly, the ability to trace artifacts back to review instances and software releases relies on the artifacts' creators to include corresponding references. Note that ETHDPR in its current form cannot enforce any conditions on the structure or content of off-chain artifacts, as their contents are inaccessible to on-chain smart contracts.

O5 *(Possible attribute claims and review consolidation methods should be use case agnostic)*: ETHDPR achieves this objective by construction. By using IPFS as an off-chain storage and distribution mechanism for artifacts and only storing CIDs on-chain, ETHDPR avoids limits on the kind or size of artifacts that would have arisen by storing them on-chain. In its current form, ETHDPR supports software claims and attestations that can be expressed as text and encoded in JSON formatted files. By utilizing the interlinking of files inherently available in IPFS, complex multimedia documents could also be included in software attestations, but their encryption for a coordinated disclosure remains as future work. It is worth reiterating that gaining this flexibility in the scope and structure of review artifacts is the result of a deliberate trade-off, the downside of which is ETHDPR's inability to enforce any rules on their content via smart contracts. Essentially, with the power to construct artifacts in any way they see fit comes the individual responsibility of maintainers and reviewers to do so sensibly.

5.6. Generalization

In addition to a proof of concept, ETHDPR also serves as one particular instance of a more general structure for a decentralized independent review application, which we explore in more detail in this section. Figure 5.5 shows an abstraction of Figure 5.2 with implementation details condensed into distinct mechanisms.

In ETHDPR, Palinodia provides the necessary identities for software as the primary object of reviews. Other systems like SmartWitness [37] or Chainiac [66] could also fulfill the same functionality for software. More generally, whatever digital object should be reviewed needs to be uniquely identifiable and referenceable both for the purpose of executing reviews as well as attaching results and attestations. In Figure 5.5 this is the main task of the *(Software) Identity Management* component. Similarly important for reviews of digital objects is an integrity-protecting binding between its identity representation and the object itself. Without such a binding, the correspondence between what reviewers examine and what relying parties obtain is not ensured, defeating the purpose of performing reviews.

Next, ETHDPR includes ETHTID as a *Disclosure Coordination Mechanism* in order to both enable the publication of independently generated review results and to make that fact evident afterwards. With front-running attacks on blockchains and corresponding defenses garnering more attention in recent years [29] and coordinated disclosure of transactions being one approach for defensive measures, distributed ledger platforms may look to include mechanisms to provide functionalities similar to ETHTID on their protocol level, like Algorand [10]. As ETHTID has shown, decentralized applications to provide disclosure coordination can be built independent of particular use cases. Thus, even consensus-layer front-running defenses on distributed ledgers may be integrated into a decentralized review application.

Tying together the author of a review with an integrity-protecting time stamp and reference to said review is the main purpose of RevLog smart contract instances in ETHDPR as its *Review Notary Mechanism.* Rather than deploying a RevLog instance per reviewer, another approach would be to deploy a single autonomous contract instance without any access control where a review author's Ethereum address is included in the emitted event. Please recall that a secondary purpose of these events is to make reviews for a given software discoverable to any interested party. Since notary services were one of the earliest use cases explored on distributed ledgers [6], multiple services exist today [25] that could be adapted for a decentralized review application with the biggest hurdle being the aforementioned discoverability of reviews for a given software release.

Lastly, the *Attribute Consolidation Mechanism* of ETHDPR is mostly left as future work. Please recall that each review consists of one or more attributions by the respective reviewer for a given software release and stated claims. However, after all such attestations are recorded and disclosed, the logical next step is to consolidate them to approximate a more complete picture of a software's characteristics. We expect two broad classes of consolidation approaches to be possible for review applications like ETHDPR:

- **Individual consolidation**: Much like we explained above for ETHDPR, the consolidation of attributes can be performed by relying parties individually, depending on their needs and requirements for a particular software release. Such a consolidation may not necessarily consider each review instance in isolation but instead include past performances of reviewers to gauge the reliability of their attributions in comparison with the attributions of others. Depending on the number of reviews and attestations to consider, a manual approach may not be feasible and software-based aids could be necessary, which of course introduces the possibility of a circular dependence: The software to support the consolidation of software attributions would have to be reviewed itself to ensure it is functioning properly and can be relied on.
- **Collective consolidation**: By contrast to the above approach, a consolidation of published attributes could also be logged on-chain to spare relying parties the burden of performing it themselves. Part of this process, and the on-chain documentation, would have to include the resolution of conflicting attributes and the overall rationale behind the resulting consolidated attributes. To avoid reliance a centralized trusted party to perform this consolidation, a transparent and decentralized mechanism would be necessary.

Note how the difference between individual and collective consolidation reflects a core trade-off in decentralized applications between individual responsibility and delegation out of necessity or convenience.

5.7. Discussion

As a proof of concept, ETHDPR demonstrates that, through the combination of appropriate decentralized applications, software reviews can be executed and persisted on distributed ledgers. By presenting a use case for software identities on distributed ledgers, ETHDPR further adds to the affirmation of our first research question from Chapter 1. As a public infrastructure, decentralized platforms like Ethereum can serve as a singular and neutral "common ground" to focus the attention of all involved parties without relying on a centralized provider. Without the ability to modify or rescind statements after they are added to a distributed ledger, it stands to reason that greater care is taken by their authors beforehand, which is rather beneficial for the use case examined in this thesis. Likewise, many conceivable attacks leave undeniable and indelible evidence behind that can be analyzed by anyone. The append-only nature and high availability of distributed ledgers, coupled with additional assumptions for the long-term availability of off-chain artifacts in our case, can also serve as a robust history for each reviewer to build a reputation.

Despite all the benefits that distributed ledgers offer, a fundamental issue remains in that technological means can enable and encourage independent work but they can hardly enforce or guarantee it. Software attributes, as we define them for this work, require a human actor to ultimately decide whether or not to attest certain characteristics to a particular release. The same human actors have the capacity to forego any disclosure coordination mechanism and just publish their findings, thereby putting the independence of subsequent reviews into question. Likewise, collusions between reviewers may become discourageable in future iterations, but they are most likely not going to be entirely preventable through technological means either. If there is an advantage to be gained from privately sharing or aligning results between reviewers during the review period, like a reduced work load, then some reviewers may do so. If done subtly enough, even the on-chain records and off-chain artifacts may not reveal that some reviewers colluded.

Regarding the disclosure of individual review results, there is also a subtle implementation detail worth emphasizing. Please recall that, during the preparation of an encrypted report for coordinated disclosure, each reviewer generates an ephemeral discrete-log key pair to perform an ECIES key derivation using a Diffie-Hellman(-Merkle) key exchange and a cryptographic hash function. While the public half of said key pair, which we labeled R, is added to a reviews plaintext header, the private half r is both a risk and a possible fallback mechanism. If an attacker were to compromise a reviewer and obtain r covertly, they would gain premature access to said reviewer's results that were propagated via IPFS in encrypted form. However, in the same way, holding on to r would give reviewers an opportunity to individually disclose their results should the disclosure coordination mechanism fail to release its decryption key on schedule. Note that r or the ETHTID decryption key are the only feasible ways to recover the symmetric key to decrypt an ETHDPR review. Even publishing the plaintext of a review payload does not help in this regard as it cannot be encrypted and compared to the published review payload without recovering the same AES key. Consequently, without r or the ETHTID decryption key, a reviewer cannot prove which results are contained in their time-stamped and encrypted review.

Lastly, it is worth addressing our decision to realize ETHDPR, and its constituent parts, in a permissionless environment. While it allowed us to avoid relying on centralized trusted parties, it also highlighted the remaining challenge of establishing Sybil-resistant identities for ETHTID council members and ETHDPR reviewers. However, by the same argument, the concepts presented in this thesis can be transferred to a permissioned environment with minimal changes. In a permissioned setting, there exists an authority that grants and potentially revokes permission for parties to actively participate. Such a gatekeeping authority is in a position to thwart or at least hamper the creation of Sybil identities and to enforce lasting consequences for repeated misbehavior.

5.7.1. Limitations & Future Work

There are two design decisions we made in ETHDPR that highlight possible improvements and subsequent challenges. First, by tightly coupling the initiation and execution of software reviews to Palinodia, particularly the transaction to register a new software binary with its on-chain identity, there can only ever be one review per release. Similar to ETHTID, this greatly simplifies the overall organization as there cannot be simultaneous reviews with potentially overlapping claims and different disclosure schedules, but it also prevents subsequent reviews with refined claims or new ones that became relevant at a later point in time. This design decision also limits the power to select attribute claims and initiate a review to maintainers. In its current form, ETHDPR does not allow any other party to contribute attribute claims based on their demands for a particular software. To address these limitations, future improvements to ETHDPR could focus on making the selection of attribute claims and initiation of reviews more flexible while avoiding the aforementioned pitfalls.

The second design decision relates to reviews and attestations being attached to binaries of a given software, i.e. the lowest level of a software identity in Palinodia. Please recall the two-tiered tree-like structure of software identities we introduced in Chapter 3 with a root software identity as the root, intermediary software identities as the middle layer, and individual binaries as the leaves of said tree. Depending on how developers and maintainers organize their software identity, i.e. how they chose to differentiate intermediary software identities, there may be significant overlap between the binaries attached to each intermediary identity. Reviewing the same or similar claims for each new release may thus be inefficient and redundant. Instead, claims and attestations could also be attached to intermediary identities based on a common code base for example. However, this raises the issue of determining when such higher level claims need to be reevaluated due to significant changes between releases.

In this thesis, we consider each software in a standalone manner, but in practice, software of significant complexity is often built with libraries or comprised of separate reusable modules and thus more tightly interrelated. A future extension to Palinodia to represent the interrelations between different software identities that we highlighted in Subsection 3.6.1 could also be transferred to ETHDPR to avoid redundant review work. For example, a library could be reviewed and attested regarding compliance with a particular standard. Software using that library could then either inherit the standard-compliance attribute directly or have a simplified review regarding the proper use of said library to receive a derived attribute. This way, the thorough and comprehensive work to determine standard-compliance would only be necessary when the library is updated and not with every release of software that includes said library.

6. Discussion & Conclusion

In Chapter 1, we stated two research questions regarding the usefulness and adaptability of distributed ledgers for the purpose of establishing and utilizing software identities. The first question asked:

> *Can distributed ledger technologies enable the establishment, management, and use of software identities for the purpose of secure software distribution without a trusted third party?*

With Palinodia, we demonstrate at the example of Ethereum that distributed ledgers are already a viable platform to establish and manage identities for software to secure their distribution and revocation. By providing a uniquely referenceable, verifiable, and highly available record in addition to a consensus-enforced and customizable access control mechanism, distributed ledgers are well-suited to this use case. As a concept, Palinodia derives its security from fundamental DLT properties and, as such, should be applicable to other ledgers besides Ethereum. We note that our implementation of Palinodia utilizes an Ethereum-specific feature, namely events, to simplify monitoring of and searching for relevant on-chain data, particularly revocations of past releases. While the details of such a functionality on other ledgers may vary, the ability to search for and obtain specific data is rather essential to the usability of any given DLT and should therefore be available. As such, Palinodia and ETHDPR as proof of concepts on Ethereum provide an affirmatory answer to the general research question with minor caveats.

In order to realize an independent review of binaries as an additional use case for software identities, we faced the problem that, at the time of writing, Ethereum does not provide a mechanism to coordinate the disclosure of on-chain statements. Transactions submitted to the Ethereum peer-to-peer network are disseminated quickly before eventually being added to a block. Afterwards, it is not evident from the on-chain record which transactions were created independently, i.e. without each sender knowing about the other transactions, even if they end up in the same block. Overcoming this limitation lead to our second research question:

> *Can distributed ledger technologies support cryptographic protocols to achieve a coordinated disclosure of independent statements on a public ledger without a trusted third party?*

With ETHTID, we transferred established cryptographic protocols in the form of distributed key generation in conjunction with threshold secret sharing and recovery to the execution environment of distributed ledgers. By tasking a council with the responsibility of generating a temporally decoupled asymmetric key pair, arbitrarily many statements can be encrypted such that their disclosure depends on the release of a single cryptographic key that no single party holds before its recovery. In our proof of concept implementation, Ethereum smart contracts provide two essential features: First, through their address, they establish a unique point of reference for each execution instance to obtain the aforementioned key pair. Second, by codifying parts of the cryptographic protocol, they provide participants with means to punish demonstrable misbehavior. It is the second feature that limits the generalizability of our results to other distributed ledgers. Ethereum is currently limited to only one elliptic curve that, while not particularly well suited to our use case, was sufficient to realize our proof of concept implementation. Depending on which cryptographic operations are available on other ledgers as part of their application layer, a disclosure coordination mechanism akin to ETHTID may or may not be feasible on them. As such, we can only affirm our second research question with some reservations. However, our work provides a set of requirements a distributed ledger must fulfill in order to support one way of implementing a disclosure coordination mechanism.

Both our disclosure coordination mechanism as well as our system for independent reviews of software releases contain at their core a Sybil problem that is worth examining in more detail. It is instructive to briefly consider the Sybil defenses present in the consensus layers of Bitcoin and Ethereum, namely Proof of Work (PoW) and Proof of Stake (PoS). In both cases, the goal is to prevent a malicious party from gaining a disproportionate amount of control by inextricably linking the operation of proposing a new block with a limited and precious resource[1], computation power for PoW and locked capital for PoS. Yet, there is a crucial difference between PoW and PoS when it comes to the significance of identities. In Bitcoin's Proof of Work, on-chain identities of miners are only significant for the collection block rewards. For the purpose of block generation, it is irrelevant how any given miner allocates their computational resources to on-chain identities.

Proof of Stake works markedly different. To give a brief overview, rather than miners running hardware to generate Proofs of Work, Proof of Stake defines *validators* who lock a certain amount of cryptocurrency, the eponymous stake, in a smart contract in order to be eligible for proposing blocks of the ledger for that very cryptocurrency. A consensus-enforced cryptographic protocol selects an eligible validator responsible for constructing and signing each new block. If a validator provably misbehaves, for example by not publishing a block on time or by including invalid transactions, their stake can be partially destroyed or redistributed. For the purpose of this discussion, the crucial point is the tight coupling between stake and on-chain identities as well as their cryptographic selection. Note that a natural person can be in control of multiple validator identities if they have the necessary capital for the stake.

Proof of Work and Proof of Stake may be viable approaches to tackling the Sybil problem present in council formation for coordinated disclosure and publishing reviews for software respectively. Extending ETHTID from one single to multiple sequential executions as sketched in Subsection 4.7.1 would enable a PoS-based Sybil defense as follows: Parties

[1] This approach was anticipated by Douceur in 2002 [27].

interested in acting as council members register with a staking contract where they also deposit a certain amount of currency as stake. For each ETHTID instance, a council can be cryptographically selected from parties registered with the staking contract. Economic rewards and punishments from an ETHTID execution can then be applied via the staking contract. While not participating in any ETHTID instance, a registered party can deregister from the staking contract to reclaim their stake and any rewards they earned. With enough registered parties and a well-balanced staking requirement, the amount of currency necessary to break the threshold of any particular ETHTID execution would be insurmountable for most parties outside of nation state attackers without preventing legitimate participation.

By contrast, Proof of Work is more suitable as a Sybil defense for ETHDPR reviews. Similar to its originally envisioned use case of spam prevention under the name Hashcash [4], each submitted review could include a nonce chosen by its author which, when hashed together with a review's CID, would produce an output hash. In order to probabilistically prove the amount of computational work expended on a given review, each review author would spend some time after finalizing their review to find a nonce that leads to an output hash as numerically small as possible before logging their review on-chain as described in Chapter 5. Unlike Bitcoin, where there is a protocol-defined global difficulty that a block's Proof of Work must meet to be valid, it could be left to relying parties to set their individual PoW difficulty during review aggregation or use the attached Proofs of Work for weighing reviews against each other. This way, spamming multiple reviews to influence relying parties would require substantial computational resources without preventing legitimate participation. There are two noteworthy benefits of this construction compared to the way PoW is used in Bitcoin: First, preparing reviews for software releases is an occasional and temporally bounded task compared to the continuous operation of Bitcoin miners. Second, and more importantly, all computational effort spent on reviews is relevant whereas the work by all Bitcoin miners except for the lucky winner who proposes a valid new block first is essentially wasted. These two factors alleviate PoW's preeminent downside, i.e. its environmental impact [45], at least for this application.

While Proof of Work and Proof of Stake are commonly regarded as interchangeable consensus mechanisms in the context of distributed ledgers, their use as Sybil defenses sketched above suggest that there are subtle differences in their applicability to other contexts that may be worth investigating further. Proof of Stake requires a clear and fixed association between identities and their stake, thus lending itself to applications like ETHTID where discrete identities are required. Proof of Work, by contrast, sidesteps identities to some extent by focusing solely on demonstrating computational work that can be allocated and shifted between identities easily.

Another point to consider is that PoW and PoS are not mutually exclusive for applications such as those examined in this work. In the case of the PoS-based construction above, potential council members registered with a staking contract could also be required to provide a Proof of Work when joining the council of a new ETHTID instance. The difficulty of this PoW could, for example, be determined by the numerical difference between the addresses of the ETHTID instance and the registered member. As with all Sybil defense mechanisms, it is crucial to strike a balance between allowing legitimate and sensible participation while encumbering malicious actors. Whether such a combined approach is practical and beneficial remains as future work.

Independent of improvements to practical implementations, there is also the concept of software identities that can be expanded further. Please recall that the goal for this work, next to elucidating and concretizing the concept itself, was to use software identities to augment and secure the distribution and inventory of software. Put differently, software identities as we define them allow users to obtain binaries with authenticity, integrity, and the endorsement of their respective creators, which can be rescinded through revocations. However, software is created and distributed in order to be *used.* A logical next step is therefore to extend the concept of software identities to also encompass the complexities of practical software use. For the purpose of this discussion, we explore two aspects, namely variants and configurations.

In fields like automotive software, which we briefly touched on with the illustrative ETHDPR example in Subsection 5.4.1, a software release can consist of several variants for different car models or feature sets, for example. Essentially, variants are a more fine-grained subdivision of versioned binaries that we used as the lowest layer of software identities. Keeping track of the resulting complexity is already an area of current research [70]. In order for software identities to be deployed in contexts such as the automotive software industry, they must also be able to cope with variants. One particular challenge that arises in conjunction with distributed ledgers is the increased amount of information that must be stored and disseminated to secure the distribution of variants for a particular software release. The naive approach of storing hashes for all variants on-chain would more than likely run into high costs on public ledgers and scalability problems in general. However, a layer of indirection using Merkle trees, distributed hash tables, or other forms of verifiable off-chain storage may be feasible to secure the distribution of variants while retaining a distributed ledger as a trust anchor. Sidetree[2] is a specification by the Decentralized Identity Foundation that is currently exploring this approach.

Configurations, meanwhile, encompass the ways in which a particular software binary is being executed. They are particularly important for long-running software on servers or appliances where they can influence a software's security. Revocations of released binaries could be made more nuanced through the incorporation of configurations. For example, by including information regarding vulnerable configurations in a revocation, users of the affected software, or rather a client running on their systems, could determine whether the revocation is relevant to them based on the configuration the software in question is currently running. Another way to utilize configurations in software identities could be for developers or maintainers to provide ready-to-use configurations for their software and log them similar to binaries so that users can obtain both with the same guarantees.

It is worth exploring the long-term ramifications that applications such as the ones presented in this thesis could have. Assuming that distributed ledgers are increasingly

[2] `https://identity.foundation/sidetree/spec/`

employed as verifiable data sources for decentralized applications that average users interact with, potential issues on both the consumer and provider side come to mind. For consumers, i.e. applications a user wishes to access, to obtain data from a distributed ledger, a corresponding client would have to be available. While each application could bundle in an appropriate distributed ledger client and have it run in a light synchronization mode, it would be more efficient to run one instance per ledger client on the operating system level so that multiple applications can obtain chain data without a redundant use of storage and bandwidth. Including such clients as system services would also take care of potential usability problems in selecting, configuring, and maintaining them by firmly placing them into the responsibility of operating system providers. Ideally, the ability to obtain data from distributed ledgers would become as reliable, convenient, and omnipresent as lookups in the Domain Name System (DNS) are today.

On the provider side of ledger data, the capabilities to satisfy the growing number of read requests that an increasing popularity of decentralized applications would cause also needs to be addressed. The full node and light client paradigm that we describe in Subsection 2.2.1 defines two factions between which tensions may arise as the aforementioned read demand grows. Full nodes forming the peer-to-peer network of a given ledger are implicitly incentivized to tightly interconnect with each other in order to obtain and forward new ledger data efficiently. Light clients, however, are pure beneficiaries of full nodes as they exclusively request ledger data but never provide anything to full nodes in return. As a consequence, full nodes would allocate only little, if any, of their limited bandwidth and computational resources to serving light clients. If light clients become ubiquitous as we project above, the currently present resource allocation dilemma would severely limit a given ledger's ability to support decentralized applications. Similar to the practice of establishing mirrors for popular repositories like the Comprehensive TEX Archive Network (CTAN)[3], it may be in the public interest to establish and communally fund light-client-friendly full nodes for popular distributed ledgers to facilitate reliable and decentralized access.

Looking closer at Ethereum, there are several future developments planned that could address some of the general points and potential issues sketched above. During the writing of this thesis, Ethereum successfully executed "The Merge" and transitioned its consensus layer from Proof of Work to Proof of Stake in September 2022. While there are several interrelated updates on the roadmap for Ethereum, for the purpose of this discussion, we highlight only two, namely *sharding* and *proposer builder separation*.

The core idea of sharding is to separate the state of a given ledger into disjoint substates in order to reduce the burden on full nodes by having them deal with only one or a few shards rather than the whole state. Keeping shards consistent while allowing cross-shard transactions is among the challenges currently being worked on. Assuming that sharding in Ethereum becomes possible, proponents argue that a lower barrier to run a full node, albeit for just some shards, will lead to more nodes being run. Ideally, an increased number of nodes would help the Ethereum peer-to-peer network deal with the aforementioned

[3] `https://www.ctan.org/mirrors/`

increase in demand for chain data due to decentralized applications, such as those presented in this work. Whether or not the previously described disadvantaged position of light clients will be affected by sharding remains to be seen.

The construction of blocks out of pending transactions that we describe in Subsection 2.2.3 has more subtleties in practice. Broadly speaking, through the careful selection and ordering of transactions, coupled with the ability of miners/validators to insert their own transactions into blocks they are currently building, economic value can be extracted in the form of fees and arbitrage. Without countermeasures, this Miner/Maximal Extractable Value (MEV)[4] can lead to a centralization of power as wealthy miners or validators can extract more MEV than those with less capital. Through projects like MEV-boost[5], the task of building blocks and proposing them can already be separated in Ethereum. Ethereum developers currently aim to realize Proposer Builder Separation (PBS) on the protocol level as part of "The Splurge", one of four long-term update paths enabled by The Merge. Essentially, builders compete with each other to construct lucrative blocks and presenting them to proposers with a bid. Proposers, i.e. Proof of Stake validators, then select the block with the highest bid to propose to the network. There are currently several open questions regarding the details of PBS, many of which revolve around technical means to ensure incentive-compatibility of the overall construction, i.e. that it is most profitable for all involved parties to adhere to the protocol. One particular challenge in this separation of duties is the handling of knowledge, more specifically the commitment to and later reveal of particular data. A disclosure coordination mechanism like ETHTID could be useful in PBS or it may be more efficient to implement such a mechanism on the protocol level as well which could then be leveraged for applications like ETHDPR. However, depending on how a disclosure coordination mechanism is realized on the protocol level of Ethereum, it may present a trade-off between schedule flexibility and cost compared to ETHTID.

Before ending this discussion, a few circular points are worth addressing. In Chapter 1, we noted the symbiotic and bidirectional relationship between decentralized platforms and the applications they enable. This, of course, not only applies to Ethereum but to software in general as well. Projecting the notion of software identities into a possible future, operating systems and their provided services or libraries could also have referenceable identities. Requirements of software applications regarding their execution environments can then also be explicitly noted via attributes attached to their respective identity by their creators.

A very fundamental assumption when using smart contracts to mediate processes between mutually distrusting parties is that they work as intended and that their creator did not include deceitful or malicious functionality. We raise and discuss this point in more detail in Chapter 4. Seeing how enabling independent reviews of software binaries for the purpose of uncovering such functionalities, among other goals, is the focus of Chapter 5,

[4] To the best of our knowledge, the term was coined within the context of Proof of Work [22], hence "Miner [...]", and the established abbreviation was then retrofitted to be applicable to Proof of Stake and potentially other consensus mechanisms that do not rely on miners.

[5] `https://boost.flashbots.net/`

it is conceivable to subject smart contract code to such reviews via ETHDPR, for example, before deployment. Formal methods can be a cornerstone of such reviews, which we investigated in a collaboration with Schiffl *et al.* [78].

6.1. Conclusion

In this dissertation, we examined the capabilities of distributed ledgers at the example of Ethereum regarding their viability to establish and utilize identities for software. Through designing and implementing three ultimately interrelated decentralized applications, we provide both a quantitative cost estimation and semi-formal security arguments showing how desired application features can be derived from native DLT properties.

With Palinodia, we demonstrated that Ethereum in conjunction with an appropriate user client can perform the function of an authentication server to secure the distribution of binaries. Furthermore, Palinodia enables a reliable, persistent, and precise revocation for individual binaries without relying on a trusted third party. To overcome a functional limitation of Ethereum and facilitate an independent review process, we devised and implemented ETHTID to enable coordinated threshold information disclosure. By leveraging existing, albeit suboptimal, elliptic curve functionality on Ethereum, we decentralized the responsibility to reveal the contents of appropriately prepared messages to a council of configurable size. We note that ETHTID may be of independent interest in other use cases that require a decentralized yet coordinated disclosure of information. Building upon Palinodia and ETHTID, we designed and implemented ETHDPR to facilitate decentralized public reviews of attribute claims for software binaries. By utilizing IPFS for off-chain file distribution, we circumvented high on-chain storage costs while retaining the ability to log time-stamped integrity-protecting references to documents on-chain. Through this decoupling, ETHDPR does not constrain tools or methods that can be used for evaluation.

In ETHTID and ETHDPR, we encountered two variants of a problem that appears fundamental to decentralized applications regarding the independence of active parties: For two identities in an ETHTID council, it appears difficult to ensure that they actually represent and are under the control of two independent parties, i.e. the classic Sybil problem as stated by Douceur [27]. For reviewers in an ETHDPR execution, it is even more difficult, if not entirely impossible through technological means, to ensure that they perform reviews independently. Particularly the latter observation points to a rather profound fact that is often overlooked. Distributed ledger technologies are a communication technology that are embedded in, influence, and rely upon a social structure that maintains and uses them, much like the telegraph, radio, landline phone, mobile phone, and the Internet before them. Since previous communication technologies have changed the ways in which humans communicate and interact, it stands to reason that distributed ledgers may also have a similar impact once they become more widely relevant and usable. Understanding the long-term impact DLTs may have on society at large will be an interesting cross-disciplinary effort between computer science, economics, and the social sciences.

Bibliography

[1] John Adler et al. “Astraea: A Decentralized Blockchain Oracle”. In: *IEEE International Conference on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data*. IEEE, July 2018, pp. 1145–1152.

[2] Monika di Angelo and Gernot Salzer. “A Survey of Tools for Analyzing Ethereum Smart Contracts”. In: *IEEE International Conference on Decentralized Applications and Infrastructures*. IEEE, 2019, pp. 69–78.

[3] C Asmuth and J Bloom. “A modular approach to key safeguarding”. In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 208–210.

[4] Adam Back. *Hashcash - A Denial of Service Counter-Measure*. Tech. rep. 2002.

[5] Paulo S L M Barreto and Michael Naehrig. “Pairing-Friendly Elliptic Curves of Prime Order”. In: *Selected Areas in Cryptography*. Ed. by Bart Preneel and Stafford Tavares. Berlin, Heidelberg: Springer, 2006, pp. 319–331.

[6] Massimo Bartoletti and Livio Pompianu. “An empirical analysis of smart contracts: platforms, applications, and design patterns”. In: *Financial Cryptography and Data Security*. Ed. by Michael Brenner et al. Cham: Springer International Publishing, 2017, pp. 494–509.

[7] Mustafa Al-Bassam and Sarah Meiklejohn. “Contour: A Practical System for Binary Transparency”. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Ed. by Joaquin Garcia-Alfaro et al. Cham: Springer International Publishing, 2018, pp. 94–110.

[8] Patrick Baudin et al. *ACSL: ANSI/ISO C Specification Language*. Version 1.17. CEA LIST and INRIA. 2021.

[9] Patrick Baudin et al. *WP Plug-in Manual*. FRAMA-C 24.0 (Chromium). CEA LIST. 2021.

[10] Fabrice Benhamouda et al. “Can a Public Blockchain Keep a Secret?” In: *Theory of Cryptography*. Ed. by Rafael Pass and Krzysztof Pietrzak. Cham: Springer International Publishing, 2020, pp. 260–290.

[11] George Robert Blakley. “Safeguarding cryptographic keys”. In: *International Workshop on Managing Requirements Knowledge*. Los Alamitos, CA, USA: IEEE, 1979, pp. 313–318.

[12] Burton Howard Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[13] Dan Boneh and Matthew Franklin. “Efficient generation of shared RSA keys”. In: *Advances in Cryptology*. Ed. by Burton S Kaliski. LNCS. Berlin, Heidelberg: Springer, 1997, pp. 425–439.

[14] Dan Boneh, Ben Lynn, and Hovav Shacham. “Short Signatures from the Weil Pairing”. In: *Journal of Cryptology* 17 (2004), pp. 297–319.

[15] Dan Boneh et al. “Aggregate and Verifiably Encrypted Signatures from Bilinear Maps”. In: *Advances in Cryptology*. Ed. by Eli Biham. Berlin, Heidelberg: Springer, 2003, pp. 416–432.

[16] Alessandro Brighente, Mauro Conti, and Izza Sadaf. “Tell Me How You Re-Charge, I Will Tell You Where You Drove To: Electric Vehicles Profiling Based on Charging-Current Demand”. In: *European Symposium on Research in Computer Security*. Ed. by Elisa Bertino, Haya Shulman, and Michael Waidner. Cham: Springer International Publishing, 2021, pp. 651–667.

[17] Vitalik Buterin. *A next-generation smart contract and decentralized application platform*. white paper. 2014.

[18] Jan Camenisch and Markus Stadler. *Proof systems for general statements about discrete logarithms*. Tech. rep. 260. Zürich: ETH Zürich, Department of Computer Science, 1997.

[19] J Cappos et al. “A look in the mirror: Attacks on package managers”. In: *ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2008, pp. 565–574.

[20] Julien Cathalo, Benoît Libert, and Jean-Jacques Quisquater. “Efficient and Non-interactive Timed-Release Encryption”. In: *Information and Communications Security*. Ed. by Sihan Qing et al. Berlin, Heidelberg: Springer, 2005, pp. 291–303.

[21] David Chaum and Torben Pryds Pedersen. “Wallet Databases with Observers”. In: *Advances in Cryptology*. Ed. by Ernest F Brickell. Berlin, Heidelberg: Springer, 1992, pp. 89–105.

[22] Philip Daian et al. “Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability”. In: *IEEE Symposium on Security and Privacy*. 2020, pp. 910–927. DOI: 10.1109/SP40000.2020.00040.

[23] Ewen Denney and Bernd Fischer. “Software Certification and Software Certificate Management Systems”. In: *IEEE/ACM International Conference on Automated Software Engineering, Workshop on Software Certificate Management*. New York, NY, USA: ACM, 2005.

[24] Giovanni Di Crescenzo, Rafail Ostrovsky, and Sivaramakrishnan Rajagopalan. “Conditional Oblivious Transfer and Timed-Release Encryption”. In: *Advances in Cryptology*. Ed. by Jacques Stern. Berlin, Heidelberg: Springer, 1999, pp. 74–89.

[25] Damiano Di Francesco Maesa and Paolo Mori. “Blockchain 3.0 applications survey”. In: *Journal of Parallel and Distributed Computing* 138 (2020), pp. 99–114.

[26] W Diffie and M Hellman. “New directions in cryptography”. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[27] John R Douceur. “The Sybil Attack”. In: *Peer-to-Peer Systems.* Berlin, Heidelberg: Springer, Mar. 2002, pp. 251–260.

[28] Taher Elgamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE Transactions on Information Theory* 31.4 (1985), pp. 469–472.

[29] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. “SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain”. In: *Financial Cryptography and Data Security.* Ed. by Andrea Bracciali et al. Vol. 11599. LNCS. Cham: Springer, 2020, pp. 170–189.

[30] Paul Feldman. “A Practical Scheme for Non-interactive Verifiable Secret Sharing”. In: *Annual Symposium on Foundations of Computer Science.* 1987, pp. 427–438.

[31] MD Sadek Ferdous, Farida Chowdhury, and Madini O Alassafi. “In Search of Self-Sovereign Identity Leveraging Blockchain Technology”. In: *IEEE Access* 7 (2019), pp. 103059–103079.

[32] Amos Fiat and Adi Shamir. “How to Prove Yourself - Practical Solutions to Identification and Signature Problems”. In: *Advances in Cryptology.* Vol. 263. LNCS. Berlin, Heidelberg: Springer, 1986, pp. 186–194.

[33] Sebastian Friebe, Ingo Sobik, and Martina Zitterbart. “DecentID: Decentralized and Privacy-Preserving Identity Storage System Using Smart Contracts”. In: *IEEE International Conference On Trust, Security And Privacy In Computing And Communications, International Conference On Big Data Science And Engineering.* IEEE, 2018, pp. 37–42.

[34] Sebastian Friebe et al. “Coupling Smart Contracts: A Comparative Case Study”. In: *Conference on Blockchain Research Applications for Innovative Networks and Services.* IEEE, 2021, pp. 137–144.

[35] Rosario Gennaro et al. “Secure Distributed Key Generation for Discrete-Log Based Cryptosystems”. In: *Journal of Cryptology* 20.1 (2007), pp. 51–83.

[36] Rosario Gennaro et al. “Secure distributed key generation for discrete-log based cryptosystems”. In: *Financial Cryptography and Data Security.* Ed. by Jacques Stern. Vol. 1592. LNCS. Berlin, Heidelberg: Springer, 1999, pp. 295–310.

[37] Juan Guarnizo, Bithin Alangot, and Pawel Szalachowski. “SmartWitness: A Proactive Software Transparency System using Smart Contracts”. In: *ACM International Symposium on Blockchain and Secure Critical Infrastructure.* BSCI. New York, NY, USA: ACM, 2020, pp. 117–129.

[38] Diksha Gupta, Jared Saia, and Maxwell Young. “Bankrupting Sybil Despite Churn”. In: *IEEE International Conference on Distributed Computing Systems.* IEEE, 2021, pp. 425–437.

[39] Diksha Gupta, Jared Saia, and Maxwell Young. “Peace Through Superior Puzzling: An Asymmetric Sybil Defense”. In: *IEEE International Parallel and Distributed Processing Symposium.* IEEE, 2019, pp. 1083–1094.

[40] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography.* Springer Science & Business Media, 2006.

[41] Petra Heck, Martijn Klabbers, and Marko van Eekelen. “A software product certification model”. In: *Software Quality Journal* 18 (2009), pp. 37–55.

[42] Jonathan Heiss, Jacob Eberhardt, and Stefan Tai. “From Oracles to Trustworthy Data On-Chaining Systems”. In: *IEEE International Conference on Blockchain*. IEEE, 2019, pp. 496–503.

[43] Sebastian Henningsen et al. “Mapping the Interplanetary Filesystem”. In: *IFIP Networking Conference*. IEEE, 2020, pp. 289–297.

[44] *ISO/IEC 24760-1: IT Security and Privacy — A framework for identity management.* Standard. ISO/IEC, 2019.

[45] Benjamin A Jones, Andrew L Goodkind, and Robert P Berrens. “Economic estimation of Bitcoin mining’s climate damages demonstrates closer resemblance to digital crude than digital gold”. In: *Scientific Reports* 12.14512 (2022).

[46] Mark P Jones. *Dealing with Evidence: The Programatica Certificate Abstraction.* Tech. rep. 2002.

[47] Kamer Kaya and Ali Aydın Selçuk. “A Verifiable Secret Sharing Scheme Based on the Chinese Remainder Theorem”. In: *Progress in Cryptology*. Ed. by Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das. Vol. 5365. LNCS. Berlin, Heidelberg: Springer, 2008, pp. 414–425.

[48] Doowon Kim et al. “The Broken Shield: Measuring Revocation Effectiveness in the Windows Code-Signing PKI”. In: *USENIX Security Symposium*. USENIX Association, 2018.

[49] Nikita Korzhitskii and Niklas Carlsson. “Revocation Statuses on the Internet”. In: *Passive and Active Measurement*. Springer, Cham, 2021, pp. 175–191.

[50] Lawrence Lessig. *Code: And Other Laws of Cyberspace, Version 2.0.* 2nd. 2006.

[51] Anthony Bahadir Lopez et al. “A Security Perspective on Battery Systems of the Internet of Things”. In: *Journal of Hardware and Systems Security* 1.2 (2017), pp. 188–199.

[52] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. “Time-Lock Puzzles in the Random Oracle Model”. In: *Advances in Cryptology*. Ed. by Phillip Rogaway. Berlin, Heidelberg: Springer, 2011, pp. 39–50.

[53] Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. “Homomorphic Time-Lock Puzzles and Applications”. In: *Advances in Cryptology*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Cham: Springer, 2019, pp. 620–649.

[54] Wenbo Mao. “Timed-Release Cryptography”. In: *Selected Areas in Cryptography*. Ed. by Serge Vaudenay and Amr M Youssef. Vol. 2259. LNCS. Berlin, Heidelberg: Springer, 2001, pp. 342–357.

[55] Deepak Maram et al. “CanDID: Can-Do Decentralized Identity with Legacy Compatibility, Sybil-Resistance, and Accountability”. In: *IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 1348–1366.

[56] Víctor Gayoso Martínez, Luis Hernández Encinas, and Carmen Sánchez Ávila. “A Survey of the Elliptic Curve Integrated Encryption Scheme”. In: *Journal of Computer Science and Engineering* 2 (2 2010).

[57] Petar Maymounkov and David Mazières. “Kademlia - A Peer-to-Peer Information System Based on the XOR Metric”. In: *International Workshop on Peer-to-Peer Systems.* Vol. 2429. LNCS. Berlin, Heidelberg: Springer, 2002, pp. 53–65.

[58] Ralph C Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Conference on the Theory and Application of Cryptographic Techniques.* Vol. 293. LNCS. Berlin, Heidelberg: Springer, 1987, pp. 369–378.

[59] Ralph C Merkle. “Secure Communications over Insecure Channels”. In: *Communications of the ACM* 21.4 (1978), pp. 294–299.

[60] Maurice Mignotte. “How to share a secret”. In: *Workshop on Cryptography.* Ed. by Thomas Beth. Vol. 149. LNCS. Berlin, Heidelberg: Springer, 1982, pp. 371–375.

[61] Alexander Mühle et al. “A survey on essential components of a self-sovereign identity”. In: *Computer Science Review* 30 (2018), pp. 80–86.

[62] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system.* white paper. 2008. URL: `https://bitcoin.org/bitcoin.pdf`.

[63] George C Necula. “Proof-Carrying Code”. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL. New York, NY, USA: ACM, 1997, pp. 106–119.

[64] George C Necula and Peter Lee. “Safe, Untrusted Agents Using Proof-Carrying Code”. In: *Mobile Agents and Security.* Vol. 1419. LNCS. Berlin, Heidelberg: Springer, 1998, pp. 61–91.

[65] Wafa Neji, Kaouther Blibech, and Narjes Ben Rajeb. “Distributed key generation protocol with a new complaint management strategy”. In: *Security and Communication Networks* 9.17 (2016), pp. 4585–4595.

[66] Kirill Nikitin et al. “CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds”. In: *USENIX Security Symposium.* USENIX Association, 2017.

[67] Marc Ohm et al. “Backstabber’s Knife Collection: A Review of Open Source Software Supply Chain Attacks”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Ed. by Clémentine Maurice et al. Vol. 12223. LNCS. Cham: Springer, 2020, pp. 23–43.

[68] Lukasz Olejnik, Steven Englehardt, and Arvind Narayanan. “Battery Status Not Included: Assessing Privacy in Web Standards”. In: *International Workshop on Privacy Engineering.* Ed. by José M del Álamo, Seda F Gürses, and Anupam Datta. San Jose, CA, USA: CEUR Workshop Proceedings, 2017, pp. 17–24.

[69] Łukasz Olejnik et al. “The Leaking Battery”. In: *Data Privacy Management, and Security Assurance.* Ed. by Joaquin Garcia-Alfaro et al. Vol. 9481. LNCS. Cham: Springer International Publishing, 2015, pp. 254–263.

[70] Stefan Otten et al. “Model-based Variant Management in Automotive Systems Engineering”. In: *International Symposium on Systems Engineering*. IEEE, 2019, pp. 1–7.

[71] Rajwinder Kaur Panesar-Walawege et al. “Characterizing the Chain of Evidence for Software Safety Cases: A Conceptual Model Based on the IEC 61508 Standard”. In: *International Conference on Software Testing, Verification and Validation*. 2010, pp. 335–344.

[72] Torben Pryds Pedersen. “A Threshold Cryptosystem without a Trusted Party”. In: *Advances in Cryptology*. Vol. 547. LNCS. Berlin, Heidelberg: Springer, 1991, pp. 522–526.

[73] Shiyue Qin et al. “Distributed secret sharing scheme based on the high-dimensional rotation paraboloid”. In: *Journal of Information Security and Applications* 58.102797 (2021).

[74] Tripti Rathee and Parvinder Singh. “A systematic literature mapping on secure identity management using blockchain technology”. In: *Journal of King Saud University - Computer and Information Sciences* 34 (8 2022), pp. 5782–5796.

[75] Peter C Rigby and Christian Bird. “Convergent Contemporary Software Peer Review Practices”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE. New York, NY, USA: ACM, 2013, pp. 202–212.

[76] Peter C Rigby, Daniel M German, and Margaret-Anne Storey. “Open Source Software Peer Review Practices: A Case Study of the Apache Server”. In: *International Conference on Software Engineering*. ICSE. New York, NY, USA: ACM, 2008, pp. 541–550.

[77] R L Rivest, A Shamir, and D A Wagner. *Time-lock puzzles and timed-release crypto*. technical report. Massachusetts Institute of Technology, 1996.

[78] Jonas Schiffl et al. “Towards Correct Smart Contracts: A Case Study on Formal Verification of Access Control”. In: *ACM Symposium on Access Control Models and Technologies*. SACMAT. New York, NY, USA: ACM, 2021, pp. 125–130.

[79] Philipp Schindler et al. *EthDKG: Distributed Key Generation with Ethereum Smart Contracts*. preprint on `https://eprint.iacr.org/2019/985`. 2019.

[80] Adi Shamir. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[81] Jiajie Shen et al. “Understanding I/O Performance of IPFS Storage: A Client’s Perspective”. In: *International Symposium on Quality of Service*. IWQoS. New York, NY, USA: ACM, 2019.

[82] Oliver Stengele, Jan Droll, and Hannes Hartenstein. “Practical Trade-Offs in Integrity Protection for Binaries via Ethereum”. In: *International Middleware Conference*. New York, NY, USA: ACM, 2020, pp. 9–10.

[83] Oliver Stengele and Hannes Hartenstein. “Atomic Information Disclosure of Off-Chained Computations Using Threshold Encryption”. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Ed. by Joaquin Garcia-Alfaro et al. Vol. 11025. LNCS. Cham: Springer International Publishing, 2018, pp. 85–93.

[84] Oliver Stengele, Christina Westermeyer, and Hannes Hartenstein. “Decentralized Review and Attestation of Software Attribute Claims”. In: *IEEE Access* 10 (2022), pp. 66694–66710.

[85] Oliver Stengele et al. “Access Control for Binary Integrity Protection using Ethereum”. In: *ACM Symposium on Access Control Models and Technologies*. SACMAT. New York, NY, USA: ACM, 2019, pp. 3–12.

[86] Oliver Stengele et al. *ETHTID: Deployable Threshold Information Disclosure on Ethereum*. preprint on `https://arxiv.org/abs/2107.01600`. 2021.

[87] Oliver Stengele et al. “ETHTID: Deployable Threshold Information Disclosure on Ethereum”. In: *International Conference on Blockchain Computing and Applications*. 2021, pp. 127–134.

[88] Nick Szabo. “Formalizing and Securing Relationships on Public Networks”. In: *First Monday* 2.9 (1997).

[89] Edward Waring. “Problems concerning interpolations”. In: *Philosophical Transactions of the Royal Society of London* 69 (1779), pp. 59–67.

[90] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger*. yellow paper on `https://ethereum.github.io/yellowpaper/paper.pdf`. 2014.

[91] David Yakira, Ido Grayevsky, and Avi Asayag. *Rational Threshold Cryptosystems*. preprint on `https://arxiv.org/abs/1901.01148`. 2019.

[92] Haoqian Zhang et al. “Flash Freezing Flash Boys: Countering Blockchain Front-Running”. In: *IEEE International Conference on Distributed Computing Systems Workshops*. IEEE, 2022, pp. 1–6.

A. Appendix

On the following pages, we provide the pseudocode excerpts from Subsection 4.5.2 in contiguous formatting.

```
t // Threshold codified in contract
members // Set of member indices
pks[] // Ephemeral public keys of other members for shadow encryption/decryption
$\mathrm{sk}_i$ // Own secret key for shadow encryption/decryption
enc_shadows[] // Encrypted shadows broadcast by other members
es[] // Contributions to encryption key of all members, including self
poly_comms[] // Commitments to polynomials of all members, including self
dec_shadows[] // Decrypted shadows necessary for generating own share
shares[] // Shares necessary for decryption key recovery
Prepare register():
    Draw $\mathrm{sk}_i$ uniformly at random from $\mathbb{Z}_p$ and store it persistently
    Generate $\mathrm{pk}_i = g^{\mathrm{sk}_i}$
    Set members ← members ∪ $i$
    Send transaction register($\mathrm{pk}_i$)
Receive register($\mathrm{pk}_j$) from $c_j$:
    Set pks[$j$] ← $\mathrm{pk}_j$
    Set members ← members ∪ $j$
Prepare distribute_shadows():
    Obtain t from ETHTID contract
    Draw $d_i$ uniformly at random from $\mathbb{Z}_p$
    Generate $e_i = g^{d_i}$
    Set es[$i$] ← $e_i$
    Draw $\{a_{i,k}\}_{k=1}^{t}$ uniformly at random from $\mathbb{Z}_p$
    Generate $\{A_{i,k} = g^{a_{i,k}}\}_{k=1}^{t}$
    Set poly_comms[$i$] ← $\{A_{i,k}\}_{k=1}^{t}$
    for $l \in$ members do // Generate shadows for other members and self
        Generate $u_{i \to l} = d_i + \sum_{k=1}^{t} a_{i,k} l^k$
        if $l = i$ then // Shadow for self
            Set dec_shadows[$i$] ← $u_{i \to l}$
        else // Shadow for other member
            Load $\mathrm{pk}_l$ ← pks[$l$]
            Generate $k_{i,l} = \mathrm{pk}_l^{\mathrm{sk}_i}$
            Generate $\overline{u_{i \to l}} = u_{i \to l} \oplus \mathrm{H}(k_{i,l} \parallel l)$
        end
    end
    Send transaction distribute_shadows($\{\overline{u_{i \to j}}\}_{j \in \mathrm{members} \setminus i}, e_i, \{A_{i,k}\}_{k=1}^{t}$)
Receive distribute_shadows($\{\overline{u_{j \to l}}\}_{l \in \mathrm{members} \setminus j}, e_j, \{A_{j,k}\}_{k=1}^{t}$) from $c_j$:
    Set enc_shadows[$j$] ← $\{\overline{u_{j \to l}}\}_{l \in \mathrm{members} \setminus j}$
    Set es[$j$] ← $e_j$
    Set poly_comms[$j$] ← $\{A_{j,k}\}_{k=1}^{t}$
    Generate $k_{i,j} = \mathrm{pk}_j^{\mathrm{sk}_i}$
    Generate $u_{j \to i} = \overline{u_{j \to i}} \oplus \mathrm{H}(k_{i,j} \parallel i)$ // Decrypt shadow for self
    if $g^{u_{j \to i}} \neq e_j \prod_{k=1}^{t} A_{j,k}^{i^k}$ then // Shadow invalid
        Call submit_dispute against $c_j$ // See Figure 4.7
    else // Shadow valid
        Set dec_shadows[$j$] ← $u_{j \to i}$
    end
```

Figure A.1.: Pseudocode from the viewpoint of council member c_i.

```
Prepare submit_dispute():
    Load $\mathrm{pk}_j \leftarrow \mathsf{pks}[j]$
    Generate $k_{i,j} = \mathrm{pk}_j^{\mathrm{sk}_i}$
    Generate $\pi(k_{i,j})$ via Figure 4.8
    Load $\{\overline{u_{j\to l}}\}_{l\in\mathsf{members}\setminus j} \leftarrow \mathsf{enc\_shadows}[j]$
    Load $e_j \leftarrow \mathsf{es}[j]$
    Load $\{A_{j,k}\}_{k=1}^{t}, k_{i,j} \leftarrow \mathsf{poly\_comms}[j]$
    Send transaction submit_dispute($\{\overline{u_{j\to l}}\}_{l\in\mathsf{members}\setminus j}, e_j, \{A_{j,k}\}_{k=1}^{t}, k_{i,j}, \pi(k_{i,j})$)
Receive submit_dispute($\{\overline{u_{j\to l}}\}_{l\in\mathsf{members}\setminus j}, e_j, \{A_{j,k}\}_{k=1}^{t}, k_{m,j}, \pi(k_{m,j})$) against $c_j$ by $c_m$:
    // ETHTID contract adjudicates validity of dispute, members process a valid dispute
       as follows
    Set $\mathsf{members} \leftarrow \mathsf{members} \setminus j$
    Delete $\mathsf{pks}[j]$
    Delete $\mathsf{enc\_shadows}[j]$
    Delete $\mathsf{dec\_shadows}[j]$
    Delete $\mathsf{es}[j]$
    Delete $\mathsf{poly\_comms}[j]$
Prepare distribute_share():
    Generate $r_i = \sum_{j\in\mathsf{members}} \mathsf{dec\_shadows}[j]$
    Set $\mathsf{shares}[i] \leftarrow r_i$
    Send transaction distribute_share($r_i$)
Receive distribute_share($r_j$) from $c_j$:
    Generate $e = \prod_{l\in\mathsf{members}} \mathsf{es}[l]$ or obtain $e$ from ETHTID contract
    for $k = 1$ to $t$ do // Combine polynomial commitments
        Generate $A_k = \prod_{l\in\mathsf{members}} A_{l,k}$ // From $\mathsf{poly\_comms}[l]$
    end
    if $g^{r_j} = e \prod_{k=1}^{t} A_k^{j^k}$ then // Share is valid
        Set $\mathsf{shares}[j] \leftarrow r_j$
        if $|\mathsf{shares}| = t + 1$ then // Enough valid shares for recovery
            Compute $d$ via Equation (4.3)
            Send transaction submit_d($d$)
        end
    end
```

Figure A.2.: Pseudocode from the viewpoint of council member c_i.