Accountable Decryption made Formal and Practical

Rujia Li, *IEEE Member,* Yuanzhao Li, *IEEE Student Member,* Qin Wang, *IEEE Member,* Sisi Duan, *IEEE Senior Member,* Qi Wang, *IEEE Member,* and Mark Ryan

Abstract—With the increasing scale and complexity of online activities, accountability, as an after-the-fact mechanism, has become an effective complementary approach to ensure system security. Decades of research have delved into the connotation of accountability. They fail, however, to achieve *practical* accountability of decryption. This paper seeks to address this gap. We consider the scenario where a client (called encryptor, her) encrypts her data and then chooses a delegate (a.k.a. decryptor, him) that stores data for her. If the decryptor initiates an illegitimate decryption on the encrypted data, there is a non-negligible probability that this behavior will be detected, thereby holding the decryptor accountable for his decryption. We make three contributions. First, we review key definitions of accountability known so far. Based on extensive investigations, we formalize new definitions of accountability specifically targeting the decryption process, denoted as *accountable decryption*, and discuss the (in)possibilities when capturing this concept. We also define the security goals in correspondence. Secondly, we present a novel Trusted Execution Environment(TEE)-assisted solution aligning with definitions. Instead of fully trusting TEE, we take a further step, making TEE work in the "trust, but verify" model where we trust TEE and use its service, but empower users (i.e., decryptors) to detect the potentially compromised state of TEEs. Thirdly, we implement a full-fledged system and conduct a series of evaluations. The results demonstrate that our solution is efficient. Even in a scenario involving $300,000$ log entries, the decryption process concludes in approximately 5.5ms, and malicious decryptors can be identified within 69ms.

Index Terms—Accountability, Decryption, Trusted Hardware

1 Introduction

Modern cryptographic systems mainly depend on preventive strategies such as passwords, access control mechanisms, and authentication protocols to ensure privacy and security. Prior to gaining access to sensitive data or performing any action that raises privacy or security concerns, individuals must demonstrate their authorization to do so. These preventive strategies can indeed offer a degree of protection against unauthorized users attempting to infiltrate the system. Nonetheless, as the scale and complexity of computer systems increase dramatically, it has become increasingly evident that relying solely on a preventive approach is insufficient. In some break-glass scenarios [1], users have to bypass the preventive strategies to access sensitive information. Consequently, accountability becomes a crucial mechanism to supplement preventive strategies, as it can identify and punish malicious individuals who breach pre-established rules after accessing sensitive data [2].

In this work, we formalize the concept of *accountable decryption* [3][4], which is an innovative approach that allows users to share the ability to access a specific piece of information, simultaneously providing evidence of whether the information has been accessed or not. We consider a scenario with three roles, as illustrated in Figure 1.(a): *encryptor*, *decryptor*, and *judge*. An encryptor is a person who provides access for a decryptor to her encrypted data and defines policies to restrain how the decryption can be conducted. A decryptor decrypts the ciphertext and recovers the original data. Each decryption operation performed by the decryptor unavoidably produces evidence of the decryption which undergoes scrutiny by a judge. The judge verifies whether the conducted decryption complies with the predefined policies. If any policy violation is detected, the judge imposes appropriate punishments on the decryptor. In this way, the decryptor is accountable for his behavior.

Bringing accountability for decryption offers many benefits. For example, in electronic surveillance [5], the law-enforcement agency seeks access to users' sensitive data from tech companies. Yet, confidential court-ordered inquiries are processed privately to ensure investigation integrity. This curtails oversight of privacy-sensitive government actions. Indeed, there are concerns about government power and privacy, as there is a risk of eavesdropping on legal citizens. Accountable decryption offers a potential solution in this case: a user (encryptor), law-enforcement agency (decryptor), and overseer (judge) work in tandem to strike a balance between investigation requirements and privacy requirements. On the one hand, accountable decryption allows law-enforcement agencies to access sensitive data in adherence to investigative orders. On the other hand, it facilitates the detection of potential abuses of the granted warrant, as overseers can verify whether law-enforcement agencies comply with the court order's limits.

Beyond electronic surveillance, the practical accountability of decryption offers a range of general benefits for many applications (cf. Section 9.2). We summarize these benefits

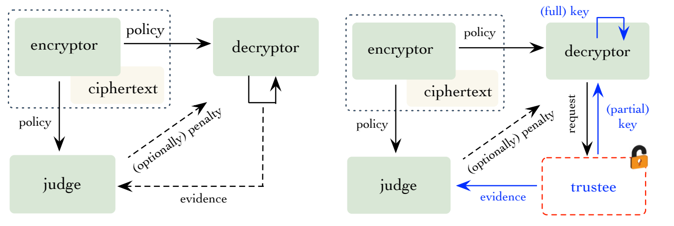
Rujia Li, and Sisi Duan are with the Tsinghua University, Beijing, China; Email: {rujia, duansisi}@tsinghua.edu.cn.
Yuanzhao Li, and Qi Wang, are with the Southern University of Science and Technology, Shenzhen, China. Email: {12232416,wangqi}@mail.sustech.edu.cn.
Qin Wang is with University of New South Wales, Sydney, Australia. Email: qinwangtech@gmail.com.
Mark Ryan is with the University of Birmingham, Birmingham, UK. Email: m.d.ryan@bham.ac.uk.

as follows.

- *Detection of unauthorized access*. Accountable decryption allows the encryptor to audit the decryption process [3].
- *Deterrents of illegal behaviours [6]*. The decryptor is responsible for its actions, as any violation of the access control to the data can be caught and punished [7].
- *Regulatory compliance*. One can also implement regulatory compliance with the pre-defined policies. The accountability of a decryptor's behavior demonstrates his compliance with regulations, as a verifiable record of decryption events is provided.
- *Key leakage awareness*. If a decryptor discovers that her ciphertexts have been decrypted without her permission, she will be alerted of a potential leakage of decryption keys.

Reviewing accountability definitions in the literature. The concept of accountability has been defined in different contexts. We reviewed accountability definitions in the literature, including those defined for identity authentication [8][9], non-tampering evidence [1][6], and policy violations [7][10]. Our observation is that while these definitions share similarities and follow the general principle of *after-the-fact behavior checking* and *misconduct blaming* [11][12], we still need a new notion for accountable decryption. In particular, prior definitions only provide a rather high-level abstraction of accountability. For instance, they fail to delineate what constitutes legitimate decryption activities. Furthermore, prior definitions often assume the existence of a trusted third party responsible for managing and tracking evidence of actions without addressing scenarios where this intermediary is compromised. Accordingly, we extend prior definitions and propose a new definition for accountable decryption.

Formalizing definitions and properties for accountable decryption. We propose new definitions of accountability that specifically target the decryption process. We emphasize two fundamental pillars for achieving ideal accountability of decryption (cf. Definition 9). First, our definitions ensure that dishonest decryptors can always be detected and punished (*non-repudiation*). Second, we ensure that honest decryptors are never wrongly accused and punished (*non-frameability*). This definition is derived from the synthesis of existing studies with various emphases. It encompasses faithful compliance at every stage, including policy setting, action performance, post-feedback, and punishments with appropriate punishments.



(a) Ideal accountable decryption (b) Practical accountab. decryption

Figure 1: General review on our solution

The above definition, however, might seem sufficient for defining the accountability of decryption. In practice, the decryption is conducted in the decryptor's local client. A decryptor could decrypt the ciphertext silently and secretly without providing decryption evidence to the judge, and thus, the misbehavior cannot be captured [4]. To achieve the requirement that the decryptor's actions are visible to the judge, we require another party called *trustee* (TS), which holds the decryption key. It receives requests from the decryptor and enforces the visibility of the requests to the judge. We show the solution in Figure 1.(b). Unfortunately, this reliance on specific roles may raise another concern for our definition, as the TS may become corrupted or untrustworthy. Thus, we propose two practical definitions for accountable decryption, considering a trusted TS (cf. Definition 11) and an untrusted TS (cf. Definition 14).

Building a practical accountable decryption scheme. We construct an innovative scheme that achieves accountable decryption, called PORTEX. Our solution leverages trusted hardware as the trustee, specifically Trusted Execution Environments (TEEs) [13][14][15]. In particular, by integrating TEEs into the trustees' infrastructure, the decryptor's private key and the audit trail of data access are securely protected by trusted hardware. Every time decryption is required, the decryptor must initiate a request to the trusted hardware. In response, the trusted hardware not only supplies the necessary decryption key but also records evidence of the transaction. This process ensures that decryption activities are accountable, thereby fulfilling the criteria outlined in Definition 11.

We have noticed that TEEs are vulnerable to fault attacks [16], software attacks [17], and side-channel attacks [18], et al. While certain countermeasures (e.g., software patches) can mitigate some of these vulnerabilities, achieving a completely secure TEE remains uncertain. Recent research [19][20] has highlighted that a compromised TEE cannot maintain the foundational security properties expected of TEE-based systems. In light of these challenges, our approach deviates from the traditional reliance on the inherent trustworthiness of TEEs. Instead, we introduce two algorithms to cope with TEE failures. The first algorithm inspired by the key management solutions used in [21], [22], is designed such that TEE only stores a partial key while the user securely holds another portion. Even if attackers compromise the TEE, they cannot deduce a complete decryption key. The second algorithm adopts a *trust, but verify* model, where we trust TEE and use its service, but empower users (i.e., decryptors) to detect the potentially compromised state of TEEs. This minimizes losses upon detecting a compromised state. By these algorithms, our system is in accordance with Definition 14. To the best of our knowledge, we proposed the first ***practical*** solution that achieves accountability in TEE-based systems without assuming absolute trust in TEEs.

Providing full-fledged implementation and evaluations. We provide a fully functional implementation[1][2] including encryption module, decryption module, tracing module, and conduct a series of evaluations for our implementation. Experimental results further demonstrate that our system is efficient. Even in an extreme scenario involving 300,000

[1]Released at https://github.com/portex-tee/portex
[2]Public service at http://ac-dec.com

log entries, the decryption process concludes in approximately 5.5ms. Furthermore, a user can identify the malicious decryptor within 69ms, while the compromised TEE is detected in a mere 4.3ms. In addition, we benchmark our proposed system against state-of-the-art studies. The results indicate that our system offers a more practical solution, as it obviates the need for a centralized storage center for ciphertext backups, thereby reducing both the number of participants and the frequency of interactions.

2 Preliminaries

Trusted hardware. We use the trusted execution environment in this work. TEE is a secure area within the main processor that operates as an isolated kernel. It ensures the confidentiality and integrity of sensitive data and computations. State-of-the-art TEE implementations include Intel Software Guard Extensions (SGX) [13], ARM TrustZone [14], RISC-V Keystone [15]. A TEE typically provides features including: *runtime isolation* and *local/remote attestation*. Without loss of generality, we use Intel SGX as the TEE instance to illustrate TEE's key features. Runtime isolation guarantees that the code execution is isolated from untrusted memory regions. Attestation proves that the application is running within trusted hardware. By following several notions from [23], we define TEE as general trusted hardware HW.

Definition 1 (Trusted hardware, HW)**.** HW *for executing a probabilistic polynomial time (PPT) program Q consists of algorithms* {HW.Setup, HW.Load, HW.Run, HW.Run&Quote, HW.VerifyQuote}.

- HW.Setup(λ): The algorithm inputs the security parameter λ, creates a secret key sk_{quote} for signing remote attestation quotes, and outputs public parameters pms_{hw}.
- HW.Load(pms_{hw}, Q): The algorithm creates an enclave and loads the code Q into the created enclave. It outputs the handle hdl of an enclave.
- HW.Run(hdl, in): The algorithm executes the program in the enclave with an input in.
- HW.Run&Quote(hdl, in): It takes as input hdl, in, and executes the program in enclaves. After obtaining the result, the enclave signs the output with sk_{quote}, and creates a quote $q = (md_{hdl}, H_Q, in, out, \sigma)$, where md_{hdl} is the enclave metadata, H_Q is a hash of Q, and σ is the signature of previous data.
- HW.VerifyQuote(pms_{hw}, q): The algorithm verifies the quote. It verifies the signature σ and outputs *true* if the verification succeeds. Otherwise, it outputs *false*.

Integrity of TEE execution. TEE creates a secure, isolated area that can run the application and handle sensitive data. This area is a self-contained execution environment completely isolated from the rest of the system. The application code cannot be replaced or modified once it is successfully loaded in this area [24]. Thus, the execution of identical loaded code with congruent inputs shall yield consistent outputs. We formalized this property as follows.

Definition 2 (TEE execution Integrity, ExeInty)**.** *Considering that an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ playing the following game.*

```
$\mathsf{G}_{\mathsf{ExeInty}}(\lambda)$
1 : $\mathcal{C}$ Run $pms_{hw} \leftarrow \mathsf{HW.Setup}(1^{\lambda})$
2 : $\mathcal{C}$ and initialize $\mathcal{O} := \emptyset$.
3 : $\mathcal{A}$ runs $hdl \leftarrow \mathsf{HW.Load}(pms_{hw}, Q)$
4 : $\mathcal{A}$ runs $q \leftarrow \mathsf{HW.Run}(hdl, in)$ outputting opt.
5 : $\mathcal{A}$ adds opt into $\mathcal{O}$
6 :  repeat step 3-5
```

The advantage $\mathsf{Adv}^{\mathsf{G}_{\mathsf{ExeInty}}}_{\mathcal{A},\mathsf{HW}}(\lambda)$ *of $\mathcal{A}$ winning the game is*

$$\mathsf{Adv}^{\mathsf{G}_{\mathsf{ExeInty}}}_{\mathcal{A},\mathsf{HW}}(\lambda) = \Pr[\mathsf{HW.Run}(hdl, in) \not\in \mathcal{O}].$$

HW *is* ExeInty *secure if for all PPT adversary $\mathcal{A}$, there exists a negligible function* negl(λ) *such that* $\mathsf{Adv}^{\mathsf{G}_{\mathsf{ExeInty}}}_{\mathcal{A},\mathsf{HW}}(\lambda) < \mathsf{negl}(\lambda)$.

Notes. In several TEE products (e.g., Intel SGX), to preserve signers' anonymity, a quote is created using an anonymous group signature [13]. As anonymity is orthogonal to our formalization, we omit it in this work.

Cryptographic primitives. Our construction relies on standard cryptographic primitives, i.e., public key encryption scheme PKE, signature scheme S (see Appendix A).

3 Definitions of Accountability

In this section, we review the development of the definitions. Our objectives are twofold: (i) to assess the applicability of the existing definitions to accountable decryption, and (ii) if they are not suitable, to identify commonalities in the current definitions that can serve as a foundation for formulating our new definition.

✎ Nissenbaum (1996) [25] first proposed the notion of accountability in computer sencece. It did not provide a clear definition of accountability but strengthened that accountability is crucial for building reliable computer science.

✎ Kailar (1996) [8] and Yumerefendi et al. (2005) [9] extended the definition of accountability by emphasizing its objectivity and establishing its strong connection to real entities. Their work emphasized the importance of binding accountability with a unique identifier, enabling the tracking and linking of actions to specific individuals or entities. This crucial exploration allows the detection of misbehavior.

Definition 3 (On linkage, merged by [8][9])**.** *An accountable system associates states and actions with identities and provides primitives for actors to validate the states and actions of their peers, such that cheating or misbehavior becomes detectable, provable, and undeniable by the perpetrator.*

✎ Subsequent endeavors have further enriched the concept of accountability by accentuating the significance of honest behaviors exhibited by participants. To achieve this, Bella et al. (2006) [1] and Haeberlen et al. (2007) [6] embrace accountability as the process of furnishing evidence to a principal, which can subsequently be presented to the judge. Buldas et al. (2000) [26] and Yumerefendi et al. (2007) [27] delved into its application and strengthened the importance of real-time detection of misconduct.

Definition 4 (On detection, rephrased)**.** *Accountability has the following features: (a) reliable evidence, merged by [1][6]: It indicates the delivery to a principal of evidence that is later* *presented to the judge. The evidence can be validated and achieves fairness (i.e., that one protocol participant gets evidence if and only if the other one does) and non-repudiation; (b) enhanced misconduct detection, merged by [26][27]: A system should provide a means to directly detect and expose misbehavior by its participants, or, enable a principal to prove to the judge any detected fraud.*

✎ A parallel line of studies, comprising Lampson (2005) [12], and Feigenbaum et al. (2011) [11], further improved the accountability definition by emphasizing that accountability is actually a posteriori behavior. These studies emphasized the significance of consequences and deterrent measures in shaping responsible behavior in various domains. Grant et al. (2005) [10] and Weitzner et al. (2008) [7] underscored the imperative of validating adherence to predefined policies.

Definition 5 (On punishment, rephrased)**.** *Accountability in a computing system implies the following properties: (a) awareness of policy violation, merged by [7][10]: Some actors have the right to hold other actors to a set of standards, to judge whether they have fulfilled their responsibilities in light of these standards. (b) posterior penalty, merged by [10][11]: An entity is accountable with respect to some policy (or accountable for obeying the policy). Whenever the entity violates the policy, with some non-negligible probability, the entity will be punished.*

✎ Subsequent studies such as Haeberlen (2010) [28] have made strides in identifying the key factors that define the connotation of accountability. In addition to that, a few studies, such as Ishai et al. (2014) [29], have delved into establishing formal treatments of accountability.

Definition 6 (A summary by [28])**.** *A system is accountable if (a) faults can be reliably detected, (b) each fault can be undeniably linked to at least one faulty node, and (c) the faulty entities will be properly sanctioned.*

The above review shows that decades of research have thoroughly studied the concept of accountability with different focuses. However, they fail to fully capture the entities, components, and algorithms involved in accountable decryption. For example, none of the definitions explicitly define the actions of encryption operations or what constitutes legitimate decryption activities. This finding highlights the need for an accountable decryption definition.

General principle of accountability. We summarize the general principle of accountability in the above definitions. It holds the following features: (1) *linkage identities*: each action is linked to an entity that performed it; (2) *reliable evidence*: the system maintains a record of past actions such that entities cannot secretly omit, falsify, or tamper; (3) *policy compliance*: the evidence can be inspected for signs of faults; (4) *detection*: when a judge detects a fault, it can obtain an alert of the fault that can be verified independently by a third party; (5) *punishment*: a proper sanction can be applied to misconduct entities.

4 Accountable Decryption

In this section, we formally define the accountability of decryption. Our definition fits in the scenario where users choose a delegate to store their encrypted data and are aware of any subsequent decryption actions of the encrypted data. The process involves multiple roles: *encryptor* (E), *decryptor* (D), and *judge* (J). E and D are entities that create ciphertexts and perform the decryption of ciphertexts. J is responsible for detecting the misbehavior and imposes penalties against D who conduct such misbehavior. Accountable decryption requires five protocols.

- ***Setup***. $params \leftarrow \mathsf{Setup}(1^\lambda)$; The system takes security parameters λ and outputs the parameters $params$. Here, $params$ is the default input for the rest of the algorithms and is thus ignored for simplicity.
- ***Encryption***. $(ct, \mathcal{P}) \leftarrow \mathsf{Enc}(aux, m)$; An encryptor E executes this algorithm to generate a ciphertext ct by using a message m and decryptor's auxiliary data aux, and policies $\mathcal{P}$. Here, $\mathcal{P}$ dictates what are legal actions.
- ***Decryption***. $(m, \pi)/bot \xleftarrow{\widetilde{e}} \mathsf{Dec}(key, ct)$; A decryptor D executes this algorithm within a designated environment, denoted as $\widetilde{e}$. It takes as input a private key key, a ciphertext ct, and the encapsulated environment variables of $\widetilde{e}$. If the decryption is successful, it outputs a plaintext m and a piece of evidence π, which serves as substantiation for validating the decryption process. Otherwise, it outputs $\perp$, indicating an abort. Notably, $\widetilde{e}$ captures critical aspects of the event, including precise timing, the unfolding sequence, and the identities of participating entities. Ideally, π faithfully represents $\widetilde{e}$, which we denote as $\pi \Leftrightarrow \widetilde{e}$.
- ***Check violation***. $tag \leftarrow \mathsf{CheckViolation}(\pi, ct, \mathcal{P})$; A judge J executes this algorithm to scrutinize the actions of the decryptor, ensuring compliance with the predefined policies. It takes as input the evidence of decryption π, the policy $\mathcal{P}$ and outputs a result $tag \in \{true, false\}$. Here, $false$ indicates the decryptor's action is aligned with policies, and we define it as legitimate decryption.
- ***Punishment***. $true/false \leftarrow \mathsf{Punish}(tag, \mathcal{P})$; J imposes penalties against D in case of non-compliance. Here, $true$ indicates penalties are imposed successfully.

Our definition follows the general principle summarized in Section 3. It has the ability to *trace*, *identify*, and *punish* malicious decryptors. In particular, if D maliciously accesses the plaintext, J provides feedback on D 's actions by checking evidence π, and then imposes penalties upon D for its misbehaved decryption.

4.1 Ideal Accountable Decryption

We formalize the ideal notion of accountable decryption in terms of two properties: *non-repudiation* which says that decryptors who did illegal decryption can be identified and penalized, and *non-frameability*, which says that decryptors who performed legal decryption should not be blamed or subjected to false accusations.

To formalize non-repudiation, we consider a game in which the adversary takes on the role of the decryptor and is allowed to interact with other honest roles. The adversary wins if it illegally decrypts a ciphertext (policy deviation, i.e., $\widetilde{e} \not\sim \mathcal{P}$) without facing punishment (the ***punishment*** protocol outputs *false*).

Definition 7 (Non-repudiation)**.** *Define* $\mathsf{adv}_{\mathcal{A}}^{\mathsf{G_{nrep}}}(\lambda) = \Pr[\mathsf{G}_{\mathcal{A}}^{\mathsf{nrep}}(\lambda)]$*, where* $\mathsf{G}_{\mathcal{A}}^{\mathsf{nrep}}(\lambda)$ *is defined as follows:*

```
$\mathsf{G}_{\mathcal{A}}^{\mathsf{nrep}}(\lambda)$
$params \leftarrow \mathsf{Setup}(1^{\lambda})$
$(ct, \mathcal{P}) \leftarrow \mathsf{Enc}(aux, m)$
$(m, \pi) \xleftarrow{\widetilde{e}} \mathcal{A}^{\mathsf{Dec}(key, ct)}$
$tag \leftarrow \mathsf{CheckViolation}(\pi, ct, \mathcal{P})$
$\mathbf{return}\ (false = \mathsf{Punish}(tag, \mathcal{P}) \wedge (\widetilde{e} \not\prec \mathcal{P}) \wedge (\pi \Leftrightarrow \widetilde{e})$
```

A protocol satisfies non-repudiation, if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\lambda)$ *satisfying* $\mathsf{adv}_{\mathcal{A}}^{\mathsf{G_{nrep}}}(\lambda) < \mathsf{negl}(\lambda)$.

Next, to formalize non-frameability, we consider an adversary aiming to frame an honest decryptor by generating evidence of its "misbehavior", to ensure the unjust punishment of the honest decryptor. We set up a game where the adversary can assume any role in the system, including that of the encryptor and the judge, and interact with an honest decryptor. The adversary wins by producing evidence of its "misbehavior" and inducing punishment upon the legal decryptor who decrypted a ciphertext in accordance with the policy i.e., $\widetilde{e} \prec \mathcal{P}$.

Definition 8 (Non-frameability)**.** *Define* $\mathsf{adv}_{\mathcal{A}}^{\mathsf{G_{nfrm}}}(\lambda) = \Pr[\mathsf{G}_{\mathcal{A}}^{\mathsf{nfrm}}(\lambda)]$*, where* $\mathsf{G}_{\mathcal{A}}^{\mathsf{nfrm}}(\lambda)$ *is defined as follows:*

```
$\mathsf{G}_{\mathcal{A}}^{\mathsf{nfrm}}(\lambda)$
$params \leftarrow \mathsf{Setup}(1^{\lambda})$
$(ct, \mathcal{P}) \leftarrow \mathsf{Enc}(aux, m)$
$(m, \pi) \xleftarrow{\widetilde{e}} \mathcal{A}^{\mathsf{Dec}(key, ct)}$
$tag \leftarrow \mathsf{CheckViolation}(\pi, ct, \mathcal{P})$
$\mathbf{return}\ (true = \mathsf{Punish}(tag, \mathcal{P}) \wedge (\widetilde{e} \prec \mathcal{P}) \wedge (\pi \Leftrightarrow \widetilde{e})$
```

A protocol satisfies non-frameability, if for all PPT adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\lambda)$ *satisfying* $\mathsf{adv}_{\mathcal{A}}^{\mathsf{G_{nfrm}}}(\lambda) < \mathsf{negl}(\lambda)$.

Definition 9 (Ideal Accountable Decryption, IAD)**.** *A system achieves* IAD *if it satisfies non-repudiation and non-frameability.*

Definition 9 seems sufficient for defining the accountability of decryption. However, in practice, decryption occurs locally. If a decryptor withholds π or provides false π (i.e., π cannot faithfully reflect $\widetilde{e}$, namely $\pi \not\Leftrightarrow \widetilde{e}$) during the decryption process, misbehaviors will never be captured.

4.2 Practical Accountable Decryption

To address this challenge, we introduce a new role to manage key and π, and we call this role *trustee* (TS). Accordingly, we introduce one protocol (called, EvidenceKeyManage) with two interactive sub-protocols.

- ***Management***. $(key', \pi') \xleftarrow{\widetilde{e_1}} \mathsf{Manage}(ct, \mathcal{P})$; A trustee TS executes this protocol, producing a key called key' and a piece of evidence π if $\widetilde{e_1} \prec \mathcal{P}$.
- ***Key generation***. $key \leftarrow \mathsf{KeyGen}(key')$; A decryptor D derives its decryption key from key'.

In the scenario above, D must obtain the key from TS before executing decryption, preventing the issue of evidence withholding. However, this brings a new problem: TS may behave maliciously. Therefore, we propose two definitions for accountable decryption, considering a trusted TS and an untrusted TS.

A trusted TS must fulfill a new requirement: TS should provide authentic keys and evidence. Formally, we define this requirement by *evd-key-soundness* game. In this game, we consider a game in which an adversary plays the role of TS and is allowed to interact with other honest roles. The adversary wins if it generates illegal evidence that cannot represent its decryption behavior ($\pi^\star \not\Leftrightarrow \widetilde{e_\star}$) or generates an illegal key that cannot decrypt the ciphertext.

Definition 10 (Evd-key-soundness)**.** *Supposing* $\mathsf{adv}_{\mathcal{A}}^{\mathsf{G_{sound}}}(\lambda) = \Pr[\mathsf{G}_{\mathcal{A}}^{\mathsf{sound}}(\lambda)]$*, where* $\mathsf{G}_{\mathcal{A}}^{\mathsf{sound}}(\lambda)$ *is defined as follows:*

```
$\mathsf{G}_{\mathcal{A}}^{\mathsf{sound}}(\lambda)$
$params \leftarrow \mathsf{Setup}(1^{\lambda})$
$(ct, \mathcal{P}) \leftarrow \mathsf{Enc}(aux, m)$
$(key^\star, \pi^\star) \xleftarrow{\widetilde{e_\star}} \mathcal{A}^{\mathsf{Manage}(ct, \mathcal{P})}$
$key \leftarrow \mathsf{KeyGen}(key^\star)$
$\mathbf{return}\ (\pi^\star \not\Leftrightarrow \widetilde{e_\star}) \vee (\perp = \mathsf{Dec}(key, ct))$
```

A protocol satisfies evd-key-soundness, if for all PPT adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\lambda)$ *satisfying* $\mathsf{adv}_{\mathcal{A}}^{\mathsf{G_{sound}}}(\lambda) < \mathsf{negl}(\lambda)$.

Definition 11 (Practical accountable decryption with trusted trustee, PAD-tTS)**.** *A system achieves* ADec-tTS *if it satisfies non-repudiation, non-frameability and evd-key-soundness.*

We now consider the worst-case scenario where TS is not fully trusted. We require two additional properties: *compromise-security*, which says that even if TS is compromised, our PAD system is still secure; the compromised TS cannot learn D's plaintext; *compromise-awareness*, which say that it has a certain probability of detecting and learning the TS status of being compromised.

In the *compromise-security* game, we consider a game in which an adversary takes on the role of TS and is allowed to interact with other honest roles. The adversary wins if it can learn D's plaintext.

Definition 12 (Compromise-security)**.** *Define* $\mathsf{adv}_{\mathcal{A}}^{\mathsf{G_{csec}}}(\lambda) = \Pr[\mathsf{G}_{\mathcal{A}}^{\mathsf{csec}}(\lambda)]$*, where* $\mathsf{G}_{\mathcal{A}}^{\mathsf{csec}}(\lambda)$ *is defined as follows:*

```
$\mathsf{G}_{\mathcal{A}}^{\mathsf{csec}}(\lambda)$
$params \leftarrow \mathsf{Setup}(1^{\lambda})$
$(ct, \mathcal{P}) \leftarrow \mathsf{Enc}(aux, m)$
$(key^\star, \_) \xleftarrow{\widetilde{e_1}} \mathcal{A}^{\mathsf{Manage}(ct, \mathcal{P})}$
$key \leftarrow \mathcal{A}^{\mathsf{KeyGen}(key^\star)}$
$\mathbf{return}\ m = \mathcal{A}^{\mathsf{Dec}(key, ct)}$
```

A protocol satisfies compromise-security, if for all PPT adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\lambda)$ *satisfying* $\mathsf{adv}_{\mathcal{A}}^{\mathsf{G_{csec}}}(\lambda) < \mathsf{negl}(\lambda)$.

We further introduce an additional detection protocol to capture *compromise-awareness*, namely, $true/false \leftarrow \mathsf{Trace}(\pi, \widetilde{e})$; The algorithm outputs $true$ only when the misbehavior that π cannot reflect $\widetilde{e}$ (namely $\pi \not\Leftrightarrow \widetilde{e}$), can be detected. We then consider a game in which an adversary takes on the role of TS and is allowed to interact with other honest roles. The adversary wins if it generates a π

which cannot reflect $\widetilde{e}$, namely $\pi \not\Leftrightarrow \widetilde{e}$, and such misbehavior cannot be detected.

Definition 13 (Compromise-awareness). *Supposing* $\mathsf{adv}_{\mathcal{A}}^{\mathsf{G}_{\mathsf{aware}}}(\lambda) = \Pr[\mathsf{G}_{\mathcal{A}}^{\mathsf{aware}}(\lambda)]$, *where* $\mathsf{G}_{\mathcal{A}}^{\mathsf{aware}}(\lambda)$ *is defined as follows:*

$$\begin{array}{l}
\mathsf{G}_{\mathcal{A}}^{\mathsf{aware}}(\lambda) \\
\hline
params \leftarrow \mathsf{Setup}(1^{\lambda}) \\
(ct, \mathcal{P}) \leftarrow \mathsf{Enc}(aux, m) \\
(key^{\star}, \pi^{\star}) \xleftarrow{\widetilde{e_{\star}}} \mathcal{A}^{\mathsf{Manage}(ct, \mathcal{P})} \\
\mathbf{return}\ (\pi^{\star} \not\Leftrightarrow \widetilde{e_{\star}}) \land (false = \mathsf{Trace}(\pi^{\star}, \widetilde{e_{\star}}))
\end{array}$$

A protocol satisfies compromise-awareness, if for all PPT adversaries $\mathcal{A}$, *there exists a negligible function* $\mathsf{negl}(\lambda)$ *satisfying* $\mathsf{adv}_{\mathcal{A}}^{\mathsf{G}_{\mathsf{aware}}}(\lambda) < \mathsf{negl}(\lambda)$.

Definition 14 (Practical accountable decryption with untrusted trustee, PAD-uTS). *A system achieves* ADec-uTS *if it satisfies compromise-security and compromise-awareness.*

5 A Secure Construction

Philosophy of our design. The main idea behind our construction is to establish a dependable party TS using TEEs [13][14][15]. Traditional methods of ensuring accountability in decryption rely heavily on a judge to verify the legality of each decryption. However, even if the judge is assumed to be fully honest, obtaining evidence is challenging if a decryptor refuses to comply with the protocol or provides falsified information. To address this issue, we introduce a new role called trustee operating within TEEs. It receives decryption requests from decryptors and ensures the visibility of these requests to the judge. By integrating TEEs, our construction improves the reliability of trustees, implementing tamper-evident measures during key distribution to enhance key security and traceability in the distribution process. We have observed that TEEs are susceptible to numerous attacks such as [16], [17], [18]. Achieving a completely secure TEE remains a challenge. In response, we propose a design that enables the detection of compromised trustees.

Strawman protocol. Our naive solution involves four steps. ① E, defines the decryption policy, encrypts a message, and sends the ciphertext to D. ② To decrypt the ciphertext, D submits a decryption request to TEE-based TS with a signature to prove the identity. ③ TEE retrieves the decryption key, generates a piece of evidence about the key request, stores it in a log, and then sends the decryption key to D. Then, D decrypts the ciphertext. ④ J checks the evidence and imposes penalties against malicious decryption. The above approach immediately achieves Definition 10 when we assume that TEE is fully trusted. Under this assumption, the codes are executed as loaded, ensuring the integrity of the execution; when a key request is made by the decryptor, both the evidence and the key are produced accurately.

Technical challenges. Definition 14 further assumes that the trustee can be compromised. Indeed, this is aligned with the fact that TEE products suffer from architectural vulnerabilities [30], side-channel attacks [31], and fault attacks [16]. Therefore, we need to provide a solution that satisfies Definition 14. However, a few technical challenges still exist in building a fully-fledged solution.

Challenge-1: How to protect decryption keys under compromised TEEs. In the above approach, if a TEE is compromised, the adversary can access private keys. This not only renders the accountable mechanism ineffective but also jeopardizes the confidentiality of ciphertexts. In fact, this issue is a major drawback of existing TEE-based accountability systems, such as those presented in [32], [4].

Inspired by the key management solutions used in [21], [22], we introduce a key-splitting mechanism wherein TEE only stores a partial key while the user holds another partial one. Even if an attacker accesses the key inside TEEs, it cannot obtain a full decryption key.

Challenge-2: How to detect the compromised TEE. Within the present design paradigms, TEE functions as an encapsulated entity (black box), precluding any interrogation of its internal states by unauthorized peripheral components. However, this attribute also engenders a notable challenge: the dearth of visibility for overseers seeking potential compromise or deviations from established protocols.

We develop a detection algorithm to find potential compromises by inspecting TEE's outputs. Our algorithm is based on the concept of deception technology [33]: using tactics to deceive malicious TEE into attacking the wrong targets and thus producing potentially useful information. To create wrong targets, we introduce two new roles: encryption inspector $\mathsf{P_e}$ and encryption inspector $\mathsf{P_d}$. They emulate conventional encryptors and decryptors, respectively. If TS outputs inconsistent results, $\mathsf{P_e}$ and $\mathsf{P_d}$ will learn potential compromise of TEE. Beyond that, by leveraging the commitment scheme, our newly proposed key generation algorithm guarantees that TS cannot generate a decryption key identical to the key requested by users. Thus, a pair of private keys, which are honestly generated under D's request and maliciously generated by TS, can serve as evidence of a fraudulent TS.

Overview of our construction. In our design, TS comprises two essential elements: *private key generator* (PKG) and *log manager* (LM). These elements can operate on the same server but within distinct domains for specific functionalities. PKG is executed within TEEs and handles the generation of decryption keys. LM operates in untrusted environments and is responsible for maintaining and updating the evidence π related to the key extraction process.

Threat model. We assume that the decryptor's public key can be linked to a unique identifier. This assumption is common in the literature and reasonable in practice. Instead of fully trusting the TEE like previous TEE-based solutions [3][4][34][32], we make TEE work in the "trust, but verify" model where a compromised state is detectable. Also, we require cryptographic primitives that we rely upon to be provably secure.

At a high level, our system studies as follows. Initially, E defines the condition (i.e., $\mathcal{P}$) of key extraction; TS runs PKG in TEE while providing public interfaces. Then, ① E encrypts a message and sends the corresponding ciphertext ct and $\mathcal{P}$ to D. ② for decrypting ct, D sends a key request message with a commitment for his random number to TS. ③ TS generates decryption keys and updates the evidence

π of key extraction. Specifically, LM first stores D's key request in a log and transforms it as evidence π, while PKG generates the partial decryption key based on the received commitment. ④ J traces log to find the misbehavior of decryption and imposes penalties against the decryptor. ⑤ the inspectors (i.e., E, D, J) identify and prove the guilty of dishonest/compromised TEE who suppressed the evidence/key and provided the forged evidence/key.

Final design. Our solution generally follows the strawman approach mentioned in Section 5. The concrete decryption scheme follows the design of identity-based encryption (IBE) [35][36]. We run PKG inside TEE, and thus, the master private key of IBE is securely protected. Every ciphertext is encrypted using the decryptor's identity and a serial number. To perform decryptions, a decryptor D has to request the key from PKG. An essential aspect of this procedure is that the request is forced to be stored in a log, and PKG issues a decryption key only if the request is proven to be securely stored. The log is structured as an append-only Merkle tree, as used, for example, in certificate transparency [37] to improve security and efficiency. Below, we describe the technical details.

(1) *Key management.* E encrypts a message using the receiver's identity (e.g. email addresses) and E-generated serial number. This brings two benefits. First, it eliminates the need for E to request a new key for every encryption operation and thus improves practicality. Second, due to the nature of IBE scheme, the dynamic serial number makes a fresh key. D never knows the serial number until he obtains the ciphertext, which forces him to request a new key each time for decryption and then leaves a piece of unique evidence. Meanwhile, we require TS to only maintain a *partial* private key. Through a local calculation process, D combines his partial key with that of TS to obtain the final private key. This design ensures that, even if an attacker fully gains access to the TEE-based PKG, it cannot deduce the complete decryption key (adhering to Definition 14.*vi*).

(2) *Evidence management.* We use an append-only Merkle tree based log $\mathcal{LOG}$ to store the decryption evidence π. Whenever a decryption occurs, trusted hardware updates the Merkle tree by extending the leaf from the rightmost branch. This unique structure enables the generation of two distinct types of proofs to ensure the integrity of the evidence: proof of presence ρ guarantees that the specific evidence is stored within a tree; the proof of extension ε proves that such a tree is an append-only extension of $\mathcal{LOG}$. We give an example to explain how to verify the ρ and ε. As shown in Figure 2, it depicts the Merkle tree before and after the insertion of node N_7. The ε is valid only if the root hash H is the hash of $\{H(6,6),\ H(4,5),\ H(0,3)\}$, and the ρ is valid only if the current root hash H' is the hash of $\{H(7,7),\ H(6,6),\ H(4,5),\ H(0,3)\}$.

(3) *Decryption checking.* Based on the decryption activities, a judge J conducts log queries, thereby acquiring supplementary details pertinent to the decryption event. Subsequently, J compares these supplementary details with predefined decryption policies to validate the legitimacy of the decryption process.

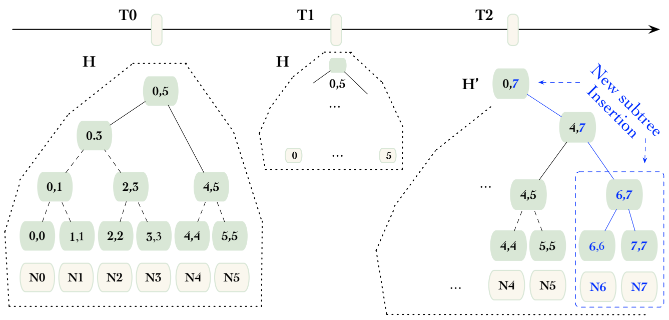


Figure 2: Merkle tree update

Our design also considered the worst-case scenario that a compromised TS may manipulate evidence and suppress evidence (adhering to Definition 14). We accordingly propose a *detection algorithm* to detect such misbehaviors. Central to this algorithm lies the deployment of *inspector*s. In a bid to mimic a conventional encryptor and decryptor, $\mathsf{P_e}$ and $\mathsf{P_d}$ dynamically engage in standard interactions with TS (behaving like a challenger). Based on the observations meticulously gathered by $\mathsf{P_e}$ and $\mathsf{P_d}$ throughout these interactions, our algorithm possesses the capability to effectively discern whether TS has been compromised.

(1) TS *provides forged evidence or forged key.* When $\mathsf{P_d}$ requests a private key, it is required to submit a signature on the identity and timestamp. Then, signatures are organized into a secure Merkle tree structure by extending the tree to the right, following an append-only policy. This empowers $\mathsf{P_d}$ to detect any forged evidence originating from TS. Meanwhile, if the issued partial key lacks the capability to yield a complete key, then $\mathsf{P_d}$ knows TS has been compromised, e.g., TS provides an invalid partial key.

(2) TS *suppresses the evidence or key.* An inspector $\mathsf{P_d}$ pretends to be a regular D and asks for a private key from TS. If the request is granted, but TS fails to show the necessary evidence or the decryption key, $\mathsf{P_d}$ knows that TS has hidden them.

We now delve into the most tricky scenario: TS effectively conceals both the key and evidence. If TS refrains from any communication with $\mathsf{P_d}$, its malicious actions may remain undetected. In our solution, the private key is generated based on $\mathsf{P_d}$'s commitment to an undisclosed random number, unbeknownst to TS. In the event that a decryptor, $\mathsf{P_d}$, subsequently uncovers a new key not linked to the commitment, and no traces are left behind, it becomes evident that this key must have been clandestinely generated by TS. Therefore, for a given identity, the existence of a pair of private keys stemming from different sources acts as evidence against a deceitful TS. Admittedly, this relies on probabilistic grounds since $\mathsf{P_d}$ cannot guarantee a 100% probability of finding a malicious key. Nevertheless, it at least exposes $\mathsf{P_d}$ to the risk of being detected.

(3) TS*, privately generates key without leaving any evidence.* When $\mathsf{P_d}$ requests a private key, it is required to use a commitment to conceal a random number. Under normal circumstances, the private key should be tailored to this specific commitment. If a new key that is not associated with the commitment is later

discovered, $\mathsf{P_d}$ will learn TS must have conducted an invalid decryption.

Notably, existing TEE-based studies assume that the hardware is fully trusted. However, in practice, real-world scenarios diverge from this ideal, as even reliable hardware like TEEs exhibit vulnerabilities, as evidenced by studies such as [30][31]. Our work takes a step forward and considers accountable decryption under the bad cases of trusted hardware being compromised. Our construction is critical for building practical decryption with accountability.

6 Detailed Protocol

In this section, we present our detailed protocol (see Figure 3). We instantiate TEE using Intel SGX [13].

Setup the system. ✎ Portex.Setup(λ): This is a preparation phase aiming to set up the system. The algorithm takes as input a security parameter λ, and outputs public parameters pms. In particular, it first calls HW.Setup to initiate the HW and generates pms_{hw} ($L.1$). Then, it runs HW.Load to load PKG code (e.g., Portex.EvidenceKeyManage as defined in Figure 3) into a generation enclave GE ($L.2$). Next, it runs ("$init$", λ) to set up the enclave and then publish the public keys (mpk, vk_{GE}) ($L.3$). Meanwhile, the log manager LM calls MT.Init(λ) to initialize the log and runs S.KGen(λ) to generate a signature key pair (vk_{LM}, sk_{LM}) ($L.4$). For D, it generate a key pair $(pk_{cli}^{enc}, sk_{cli}^{enc})$ and signing key pair $(vk_{cli}^{sign}, sk_{cli}^{sign})$ for secure data transfer ($L.5$-$L.6$). Here, parameters $(pms_{hw}, mpk, vk_{LM/GE}, pk_{cli}^{enc}, vk_{cli}^{sign})$ are publicly accessible, and default inputs of the rest of algorithms.

① **The encryption phase.** ✎ Portex.Enc(ID, m): In this phase, E encrypts a secret/message m and generates a ciphertext ct and corresponding policy $\mathcal{P}$. In particular, it first generates a random serial number SN. Then, it runs IBE.Enc($mpk, ID|SN, m$) to encrypt a message m using mpk and a combination of user identity ID and SN ($L.2$).

② **The decryption phase.** A decryptor D recovers the plaintext, before which he needs to request the corresponding private key. This is an interactive protocol between TS (including KM, LM), and D with two algorithms as follows. ✎ Portex.EvidenceKeyManage(ID, SN, σ_{cli}): The algorithm takes as input ID, SN, and D's signature σ_{cli}, and outputs a decryption key $pkey$, and the evidence π of key extraction. Inspired by [36], we separate the key generation into three sub-algorithms {IBE.KGen$_{D1}$, IBE.KGen$_{PKG}$, IBE.KGen$_{D2}$}. The detailed algorithms are as follows.

- ($L.1$-$L.3$): D first locally runs IBE.KGen$_{D1}$ to select (t_0, θ) and to generate a commitment C. Then, it generates a signature σ_{cli} on ID and SN. After that, it sends (ID, SN, C), and σ_{cli} to LM.
- ($L.5$-$L.7$): LM runs MT.Insert. If the verification of σ_{cli} fails, the algorithm aborts. Otherwise, it runs MT.Insert adds a tuple $N(ID, SN, \tau, \sigma_{cli})$ to an append-only log $\mathcal{LOG}$. To be specific, LM first records the timestamp τ of the D's request. Then, it calls MT.Insert and outputs evidence π, which contains the newly inserted tuple N, current tree root hash H_{new}, old tree root hash H_{old}, proof of presence ρ, proof of extension ε. Next, LM signs a tuple $ir(\pi, C)$ by calling S.Sign(sk_{LM}, ir) and obtains a signature σ_{ir}. Finally, LM sends (ir, σ_{ir}) to PKG.
- ($L.8.1$-$L.8.6$): Once PKG receives (ir, σ_{ir}), it calls the Portex.EvidenceKeyManage inside GE. Specifically, it first verifies σ_{ir} using vk_{LM}. If the above steps fail, the algorithm aborts. Next, GE runs MT.Verify to check the validity of the proof of presence ρ and the proof of extension ε. ($L.8.7$-$L.8.11$): If the evidence π is valid, GE selects (r', t_1) from $\mathbb{Z}_p^\star$ and starts to calculate a partial decryption key $pkey$ by calling IBE.KGen$_{PKG}$, where $pkey$ consists of (d_1', d_2', d_3'), and C is used to calculate d_1'. Then, GE encrypts $pkey$ to ct_{pkey} with D's public key pk_{cli}^{enc}. PKG sends $quote$ of ct_{pkey} to D.
- ($L.9$-$L.15$): D finally runs HW.VerifyQuote to verify $quote$. If the verification fails, the algorithm aborts. Otherwise, it calculates the final decryption key by calling IBE.KGen$_{D2}$. The final decryption key is calculated from $pkey$, and consists of (d_1, d_2, d_3).

✎ Portex.Dec(key, ct): Now, D can access the secret m by runing IBE.Dec with mpk and key ($L.1$). The algorithm outputs m if it is encrypted with $ID|SN$. Otherwise, it outputs $\perp$.

③ **The checking phase.** Portex.CheckViolation($\pi, ct, \mathcal{P}$): In this phase, J examines the actions of the decryptor based on the evidence π stored in $\mathcal{LOG}$. Precisely, it finds π in $\mathcal{LOG}$, and then checks whether π satisfies $\mathcal{P}$ ($L.1$-$L.4$). If misbehavior is found, the protocol outputs *true*. Otherwise, it outputs *false*.

④ **The punishment phase.** Punish($tag, \mathcal{P}$): The judge J imposes penalties against D who has conducted malicious decryption. These penalties include measures such as deposit forfeiture and credit downgrade, which are not detailed here for brevity.

⚠ **The detection for the compromised trustee.** For identifying potential compromises of TS, we introduce two sub-algorithms: *deterministic detection algorithm* and *probabilistic detection algorithm*. The deterministic detection algorithm infers a compromised state through a challenge-response mechanism, while the probabilistic detection algorithm identifies potential compromises by comparing the final keys issued by D with those issued by TS.

Deterministic detection algorithm. We make the inspector $\mathsf{P_d}$ pretend to be a regular decryptor D (as a challenger) and ask for the private key and evidence from the trustee TS. If any inconsistency is detected, it suggests that TS has been compromised. In particular, if P generates a partial key without evidence in $\mathcal{LOG}$, TS must have forged the evidence ($L.4$ in Algorithm 1); if P's evidence is aligned with $\mathcal{LOG}$, but the partial key lacks the capability to yield P's complete key, TS has attempted to manipulate P's key ($L.7$ in Algorithm 1); if P's partial key arises but no evidence is shown, decryption evidence suppression has happened ($L.11$ in Algorithm 1); if evidence appears without P's partial key issuance, it implies decryption key concealment by TS ($L.14$ in Algorithm 1).

Probabilistic detection algorithm. We now consider the trickiest scenario, where TS hides the newly generated key as well as evidence. We argue if TS leaks any information on its misbehavior, it runs the risk of being caught and prosecuted. In the decryption phase, as (t_0, θ) are chosen randomly by P that are unknown to PKG, PKG cannot generate a

```
$\mathsf{Portex.Setup}(\lambda)$
1 :  PKG runs $pms_{hw} \leftarrow \mathsf{HW.Setup}(\lambda)$
2 :  $hdl_{GE} \leftarrow \mathsf{HW.Load}(pms_{hw}, Q_{GE})$
3 :  $mpk, vk_{GE} \leftarrow$ $\mathsf{HW.Run\&Quote}(hdl_{GE}, (\text{“}init\text{”}, \lambda))$
         1 :  $H \leftarrow 0$
         2 :  $mpk, msk \leftarrow \mathsf{IBE.Setup}(\lambda)$
         3 :  $vk_{GE}, sk_{GE} \leftarrow \mathsf{S.KGen}(\lambda)$
         4 :  return $mpk, vk_{GE}$
4 :  LM runs $vk_{LM}, sk_{LM} \leftarrow \mathsf{MT.Init}(\lambda)$
5 :  CLIENT runs $pk^{enc}_{cli}, sk^{enc}_{cli} \leftarrow \mathsf{PKE.KGen}(\lambda)$
6 :  $vk^{sig}_{cli}, sk^{sig}_{cli} \leftarrow \mathsf{S.KGen}(\lambda)$

$\mathsf{Portex.CheckViolation}(\pi, ct, \mathcal{P})$
1 :  $(ID^\star, SN^\star, \tau^\star) \leftarrow \mathcal{LOG}$
2 :  $(N, H', H, \rho, \varepsilon) \overset{parse}{\longleftarrow} \pi$
3 :  $(ID, SN, \tau, \sigma_{cli}) \overset{parse}{\longleftarrow} N$
4 :  $\pi InLog = ((ID, SN, \tau) = (ID^\star, SN^\star, \tau^\star))$
         $\wedge\, (true = \mathsf{MT.Verify}(\pi))$
         $\wedge\, (true = \mathsf{S.Verify}(vk^{sig}_{cli}, \sigma_{cli}, ID|SN))$
5 :  return $((ID, \tau) \not\vdash \mathcal{P}) \wedge (true = \pi InLog))$

$\mathsf{Portex.Trace}(key, D)$
1 :  $(d_1, d_2, d_3) \overset{parse}{\longleftarrow} key$
2 :  $(D_1, D_2, D_3) \overset{parse}{\longleftarrow} D$
3 :  return $d_3 \neq D_3$

$\mathsf{Portex.Enc}(ID, m)$
1 :  $SN \overset{\$}{\longleftarrow} \mathbb{Z}^\star_m$
2 :  $ct \leftarrow \mathsf{IBE.Enc}(mpk, ID|SN, m)$
3 :  return $ct, SN$

$\mathsf{Portex.Punish}(tag, \mathcal{P})$
1 :  $if(tag)$ impose punishment
2 :  return $tag$

$\mathsf{Portex.EvidenceKeyManage}(ID, SN)$
1 :  CLIENT runs $C \leftarrow \mathsf{IBE.KGen}_{D1}(mpk)$
2 :  $\sigma_{cli} \leftarrow \mathsf{S.Sign}(sk^{sig}_{cli}, ID|SN)$
3 :  CLIENT sends $(ID, SN, C, \sigma_{cli})$ to LM.
4 :  $if(true \neq \mathsf{S.Verify}(vk^{sig}_{cli}, \sigma_{cli}, ID|SN)), abort$
5 :  LM runs $\pi \leftarrow \mathsf{MT.Insert}(ID, SN, \tau, \sigma_{cli})$
6 :  $ir = (\pi, C), \sigma_{ir} \leftarrow \mathsf{S.Sign}(sk_{LM}, ir)$
7 :  LM sends $(ir, \sigma_{ir})$ to PKG
8 :  $quote \leftarrow$ $\mathsf{HW.Run\&Quote}(hdl_{GE}, (\text{“}Manage\text{”}, ir, \sigma_{ir}))$
         1 :  $if(true \neq \mathsf{S.Verify}(vk_{LM}, \sigma_{ir}, ir)), abort$
         2 :  $if\,(H \neq H_{old}), abort$
         3 :  $(\pi, C) \overset{parse}{\longleftarrow} ir$
         4 :  $(N, H_{new}, H_{old}, \rho, \varepsilon) \overset{parse}{\longleftarrow} \pi$
         5 :  $(ID, SN, \tau, \sigma_{cli}) \overset{parse}{\longleftarrow} N$
         6 :  $if(false = \mathsf{MT.Verify}(\pi)), abort$
         7 :  $H \leftarrow H_{new}$
         8 :  $pkey \leftarrow \mathsf{IBE.KGen}_{PKG}(msk, ID|SN, C)$
         9 :  $ct_{pkey} \leftarrow \mathsf{PKE.Enc}(pk^{enc}_{cli}, pkey)$
        10 :  $\sigma_{pkey} \leftarrow \mathsf{S.Sign}(sk_{GE}, ct_{pkey})$
        11 :  return $ct_{pkey}, \sigma_{pkey}$
9 :  CLIENT receives quote
10 :  $(md_{hdl}, H_Q, in, out(ct_{pkey}, \sigma_{pkey}), \sigma) \overset{parse}{\longleftarrow} quote$
11 :  $if\,(H_Q \neq H_{GE}), abort$
12 :  $if\,(false = \mathsf{HW.VerifyQuote}(pms_{hw}, quote)), abort$
13 :  $if\,(false \leftarrow \mathsf{S.Verify}(vk_{GE}, \sigma_{pkey}, ct_{pkey})), abort$
14 :  $pkey \leftarrow \mathsf{PKE.Dec}(sk^{enc}_{cli}, ct_{pkey})$
15 :  $key \leftarrow \mathsf{IBE.KGen}_{D2}(mpk, ID|SN, pkey)$

$\mathsf{Portex.Dec}(key, ct)$
1 :  $m \leftarrow \mathsf{IBE.Dec}\ (mpk, key, ct)$
2 :  return $m$
```

Figure 3: PORTEX protocol and generation enclave

Algorithm 1 The *detection* algorithm

```
    wrongdoing $\in \{true, false\}$
1:  ---------deterministic detection---------
2:  upon receiving $\pi$ and $pkey$ from TS
3:      $key = \mathsf{IBE.KGen}_{D2}(mpk, ID, pkey)$
4:      if $\pi \notin \mathcal{LOG} \wedge \mathsf{Dec}(key, ct) \neq \perp$ then
5:          wrongdoing $= true$                ▷ TS must have forged $\pi$
6:  upon receiving $\pi$ and $pkey$ from TS
7:      if $\pi \in \mathcal{LOG} \wedge \mathsf{IBE.KGen}_{D2}(mpk, ID, pkey) = \perp$ then
8:          wrongdoing $= true$             ▷ TS must have forged $pkey$
9:  upon receiving $pkey$ from TS
10:     $key = \mathsf{IBE.KGen}_{D2}(mpk, ID, pkey)$
11:     if find_evidence$(\mathcal{LOG}) = \varnothing \wedge \mathsf{Dec}(key, ct) \neq \perp$ then
12:         wrongdoing $= true$           ▷ TS must have suppressed $\pi$
13: upon receiving $\pi$ from TS
14:     if find_key$(\pi) = \varnothing$ then
15:         wrongdoing $= true$        ▷ TS must have suppressed $pkey$
16: ---------probabilistic detection---------
17: upon finding $key'$
18:     if $key \neq key' \wedge \mathsf{Dec}(key', ct) \neq \perp$ then
19:         wrongdoing $= true$
```

decryption key that is same as key. This laid the foundation for detecting the PKG's misbehaviors. For a valid decryption key, d_3 contains the parameter t_0 that only knows to P. If any P finds another valid key but does not equal key, then it demonstrates that D is maliciously generated by TS, and thus shows TS's misbehavior.

7 SECURITY ANALYSIS

This section provides a security analysis, considering two scenarios where TEE is secure or compromised.

Security analyses under secure TEEs. Assuming that TEE and the used cryptographic primitives are secure, our construction achieves PAD-tTS defined in Definition 11. We prove the theorem by contradiction. If an adversary $\mathcal{A}$ made our construction fail to achieve properties in PAD-tTS, we could use such abilities to break our assumptions.

Theorem 1. *If* HW *is* ExeInty *secure and* S *is Existential Unforgeability under Chosen Message Attack (EUF-CMA) secure, our construction achieves* PAD-tTS *in Definition 11.*

```
$\mathsf{IBE.Setup}(\lambda, n)$
1 : select bilinear groups $(\mathbb{G}, \mathbb{G}_T, e, p)$, where the $p > 2^\lambda$
2 : $x \xleftarrow{\$} \mathbb{Z}_p^\star$
3 : $msk \leftarrow x$
4 : $X \leftarrow g^x$
5 : $g, h, Y \xleftarrow{\$} \mathbb{G}$
6 : $\mathsf{Z} \leftarrow (Z_0, Z_1, ..., Z_n) \xleftarrow{\$} \mathbb{G}^{n+1}$
7 : $mpk \leftarrow (X, Y, h, \mathsf{Z})$
8 : return $mpk, msk$
```

```
$\mathsf{IBE.KGen}_{D1}(mpk)$
1 : $t_0, \theta \xleftarrow{\$} \mathbb{Z}_p^\star$
2 : return $h^{t_0} \cdot X^\theta$
```

```
$\mathsf{IBE.KGen}_{PKG}(msk, ID, C)$
1 : $r', t_1 \xleftarrow{\$} \mathbb{Z}_p^\star$;
2 : $d_1' \leftarrow (Y \cdot C \cdot h^{t_1})^{1/x} \cdot H_{\mathsf{Z}}(ID)^{r'}$
3 : $d_2' \leftarrow X^{r'}$
4 : $d_3' \leftarrow t_1$
5 : $pkey \leftarrow (d_1', d_2', d_3')$
6 : return $pkey$
```

```
$\mathsf{IBE.KGen}_{D2}(mpk, ID, pkey)$
1 : $r'' \xleftarrow{\$} \mathbb{Z}_p^\star$
2 : $r \leftarrow r' + r''$
3 : $d_1 \leftarrow \frac{d_1'}{g^\theta} \cdot H_{\mathsf{Z}}(ID)^{r''}$
4 : $d_2 \leftarrow d_2' \cdot X^{r''}$
5 : $d_3 \leftarrow d_3' + t_1$
6 : $key \leftarrow (d_1, d_2, d_3)$
7 : checks $e(d_1, X) = e(Y, g) \cdot e(h, g)^{d_3} \cdot e(H_{\mathsf{Z}}(ID), d_2)$
8 : return $key$
```

```
$H(a, b)$
1 : $c \leftarrow a|b$
2 : return $\Pi.\mathsf{Hash}^{\mathsf{s}}(c)$
```

```
$\mathsf{IBE.Enc}(mpk, ID, m)$
1 : $s \xleftarrow{\$} \mathbb{Z}_p^\star$
2 : $C_1 \leftarrow X^s$
3 : $C_2 \leftarrow H_{\mathsf{Z}}(ID)^s$
4 : $C_3 \leftarrow e(g, h)^s$
5 : $C_4 \leftarrow m \cdot e(g, Y)^s$
6 : $ct \leftarrow (C_1, C_2, C_3, C_4)$
7 : return $ct$
```

```
$\mathsf{IBE.Dec}(mpk, key, ct)$
1 : $(d_1, d_2, d_3) \xleftarrow{parse} key$
2 : $m \leftarrow C_4 \cdot (\frac{e(C_1, d_1)}{e(C_2, d_2) \cdot C_3^{d_3}})^{-1}$
3 : return $m$
```

```
$\mathsf{MT.Init}(\lambda)$
1 : $\mathcal{LOG} \leftarrow \{\}$
2 : $vk_{LM}, sk_{LM} \leftarrow \mathsf{S.KGen}(\lambda)$
3 : return $vk_{LM}, sk_{LM}$
```

```
$\mathsf{MT.Insert}(T)$
1 : $n \leftarrow$ number of leaf nodes
2 : insert $T$ to $\mathcal{LOG}$
3 : $H_{old} \leftarrow H(0, n-1), H_{new} \leftarrow H(0, n)$
4 : $\rho \leftarrow \{$hash nodes covering $[0, n)\}$
5 : $\varepsilon \leftarrow \rho \cup H(n, n)$
6 : $\pi \leftarrow (N, H_{new}, H_{old}, \rho, \varepsilon)$
7 : return $\pi$
```

```
$\mathsf{MT.Verify}(\pi)$
1 : $(N, H_{new}, H_{old}, \rho, \varepsilon) \xleftarrow{parse} \pi$   2 : $H_{poe}, H_{pop} \leftarrow \phi$
3 : foreach $H_i \in \varepsilon$
4 :     $H_{poe} \leftarrow \Pi.\mathsf{Hash}(H_i, H_{poe})$
5 : endforeach
6 : foreach $H_j \in \rho$
7 :     $H_{pop} \leftarrow \Pi.\mathsf{Hash}(H_j, H_{pop})$
8 : endforeach
9 : return $(\Pi.\mathsf{Hash}(N) \in \varepsilon$ and $H_{poe} = H_{old}$
            and $H_{pop} = H_{new})$
```

Figure 4: IBE algorithm and evidence management algorithm

Proof. *We prove this theorem by demonstrating that the probabilities of the adversary winning the non-repudiation, non-frameability, and evd-key-soundness games are negligible. We discuss each of these in turn.*

Non-repudiation: To win $\mathsf{G}_{\mathcal{A}}^{\mathsf{nrep}}(\lambda)$ game, an adversary $\mathcal{A}$ must produce illegal evidence such that $false = \mathsf{MT.Verify}(\pi)$ after it successfully conducted decryption such that $m = \mathsf{IBE.Dec}$ (mpk, key, ct). As shown in Figure 3, when LM *cannot provide valid decryption evidence in the form of (ir, σ_{ir}). the algorithms L.8.1 and L.8.6 are programmed to abort. This further leads to the failure of algorithm L.8.8. Suppose $\mathcal{A}$ has successfully conducted a decryption. In that case, it implies that $\mathcal{A}$ has either manipulated the TEE code (e.g.,* Portex.EvidenceKeyManage *as defined in Figure 3) or provided a forged signature of $\mathcal{LOG}$ manager.,i.e σ_{ir}. However, the manipulation of the TEE code contradicts our assumption regarding the execution integrity of the TEE (*ExeInty *secure), while the forged signature breaks the security of EUF-CMA of* S.

Non-frameability: To win $\mathsf{G}_{\mathcal{A}}^{\mathsf{nfrm}}(\lambda)$ game, an adversary must produce illegal evidence for an honest decryptor's "misbehavior" to ensure its punishment. According to the algorithm L.5, we have known that Portex.CheckViolation *returns true only when all three conditions defined in L.4 returns true. However, the first and third conditions return true only $\mathcal{A}$ generates a valid signature $\sigma_{cli}^\star$ without sk_{cli}^{sig}, which can be used to break the security of EUF-CMA of* S. *The second condition returns true only if the TEE codes are manipulated, which contradicts our assumption regarding the execution integrity of the TEE (*ExeInty *secure);*

Evd-key-soundness: To win $\mathsf{G}_{\mathcal{A}}^{\mathsf{sound}}(\lambda)$ game, an adversary *must produce a tuple $(key^\star, \pi^\star)$, that makes $(\pi^\star \not\Leftrightarrow \widetilde{e_1})$ or $(\bot = \mathsf{Dec}(key, ct))$. However, both parameters imply that an adversary has provided a valid signature σ_{cli} (as shown in* EvidenceKeyManage *algorithm* L.4*, without providing valid σ_{cli}, the protocol will abort), which breaks the security of EUF-CMA of* S. □

Analysis under compromised TEE. We now consider cases of TS being compromised. We prove that even in this extreme case, our construction is still secure, and users can be aware of such compromises.

Theorem 2. *If discrete logarithm problem (DLP) assumption holds,* HW *is* ExeInty *secure and* S *is EUF-CMA secure, our construction achieves* PAD-uTS *defined in Definition 14.*

Proof. *We prove this theorem by demonstrating that the probabilities of the adversary winning the compromise-security and compromise-awareness game are negligible. We discuss each of these in turn.*

*Compromise-security: From the algorithm (*L.2*), we have known that $m = C_4 \cdot (\frac{e(C_1,d_1)}{e(C_2,d_2)\cdot C_3^{d_3}})^{-1}$. where, $d_1 = (Y \cdot h^{t_0+t_1})^{1/x} \cdot H_{\mathsf{Z}}(ID)^r$, $d_2 = X^r$, $d_3 = t_0 + t_1$. As r, t_0, t_1 is randomly selected from $\mathbb{Z}_p^\star$, even if the adversary fully controls x, it has a negligible probability of decrypting the ciphertext. Let the plaintext decrypted from ciphertext C as $m^\star$ and the original message as m, then we have the following equation.*

$$
\begin{aligned}
m^\star &= C_4 \cdot (\frac{e(C_1,d_1)}{e(C_2,d_2)\cdot C_3^{d_3}})^{-1} = \frac{C_4 \cdot e(C_2, X^r)\cdot C_3^{t_1+t'}}{e(C_1, \frac{d_1'}{g^{\theta'}} \cdot H_{\mathsf{Z}}(ID)^{r''})} \\
&= m \cdot \frac{e(g,Y)^s \cdot e(H_{\mathsf{Z}}(ID)^s, X^r)\cdot e(g,h)^{s\cdot(t_1+t')}}{e(X^s, \frac{(YRh^{t_1})^{1/x}}{g^{\theta'}} \cdot H_{\mathsf{Z}}(ID)^r)} \\
&= m \cdot \frac{e(g,Y)^s \cdot e(g,h)^{s\cdot(t_1+t')}\cdot e(X^r, H_{\mathsf{Z}}(ID)^s)}{e(g^{xs}, (\frac{YRh^{t_1}}{g^{x\theta}})^{\frac{1}{x}}) \cdot e(X^s, H_{\mathsf{Z}}(ID)^r)} \\
&= m \cdot \frac{e(g,Y)^s \cdot e(g, h^{t'}h^{t_1})^s}{e(g^s, \frac{Yh^{t_1}h^{t_0}\cdot X^\theta}{X^{\theta'}})} \\
&= m \cdot e(g^s, h^{(t'-t_0)} \cdot X^{(\theta'-\theta)})
\end{aligned}
$$

When $t' = t_0$ and $\theta' = \theta$ the output value is $m^* = m$. The adversary's sole source of information regarding t_0 and θ is represented by $R = h^{t_0} \cdot X^\theta = h^{t_0} \cdot g^{\theta x}$, where the public parameters g and h are prime numbers. This can be equivalently reframed as a challenge in solving discrete logarithm problems.

Compromise-awareness: We consider two cases: (Case-1) TS *provides inconsistent or forged results, and (Case-2)* TS *does not provide any result at all, i.e.,* TS *simultaneously suppresses the evidence and associated keys.*

Case-1: The proof is straightforward. If TS *fails to provide the private key or the corresponding evidence, an inspector* P *immediately learns that* TS *is compromised.*

Case-2: Given a decryption key $d = (d_1, d_2, d_3)$ generated by Portex.EvidenceKeyManage*, where $d_1 = (Y \cdot h^{t_0+t_1})^{1/x} \cdot H_{\mathsf{Z}}(ID)^r$, $d_2 = X^r$, and $d_3 = t_0 + t_1$ for distinct $t_0, t_1 \xleftarrow{\$} \mathbb{Z}_p^\star$. The* PKG *lacks information about t_0, which implies that the probability of the leaked decryption key D containing the same d_3 as key is $\frac{1}{p}$, where $p > 2^\lambda$ (as defined in* IBE.Setup*). Consequently, the probability $\mathsf{Pr}^{\mathcal{A}}(\lambda)$ of the $\mathcal{A}$'s success is:*

$$
\mathsf{Pr}^{\mathcal{A}}(\lambda) = \mathsf{Pr}[key.d_3 = D.d_3] \le \frac{1}{p} \le \frac{1}{2^\lambda}
$$

□

8 Implementation and Evaluation

We implement Portex in C++ and use Intel SGX SDK [24] as the TEE. The implementations of IBE algorithms, including key generation, encryption, decryption, and trace, are based on the Pairing-Based Cryptography (PBC) library [38]. Our Merkle Tree implementation is based on the merklecpp library [39]. The public key encryption and signature algorithms are based on the OpenSSL library [40]. The SGX Enclave does not support the original OpenSSL. Therefore, in the PKG program, we utilize SGXSSL as an alternative [41].

Performance&scalability. It involves two aspects: (a) evaluating decryption performance, misbehavior-checking performance, and detection performance for finding the compromised TEE; (b) examining how scalability is affected when the number of decryptors increases. We evaluate the performance of Portex on a machine with 3.5GHz Intel Xeon CPU (Ice Lake). The symmetric bilinear pairing $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is constructed on the curve $y^2 = x^3 + x$, and the security parameter λ is selected as 512. We report the average time of each algorithm by repeating the experiments 1,000 times.

Performance evaluation with static parameters. We first evaluate performance with static parameters. As an example, we consider the following scenario. We assume that there are 1000 decryptors, and each decryptor can launch one decryption. Our evaluation shows that the full decryption process, including two sub-algorithms: *evidence-key management* and *decryption*, concludes within 10ms. In particular, evidence-key management takes approximately 1.31ms, 3.15ms, 2.56ms to finish IBE.KGen$_{D1}$, IBE.KGen$_{D2}$, and IBE.KGen$_{PKG}$, respectively; It takes about 1.69ms and 6.05ms to generate and verify the decryption evidence. Simultaneously, the decryption of the ciphertext exhibits remarkable efficiency, demanding less than 1ms: the Dec.Setup operation concludes within an incredibly short span of 0.1ms, while IBE.Dec finalizes in a mere 0.6ms.

Meanwhile, tracing malicious decryption is significantly efficient, as it only takes at most 0.002ms to trace the evidence of malicious decryption. The process of identifying potential forgery within the TEE unfolds in two stages. Specifically, detecting forged evidence and forged keys takes approximately 0.02 milliseconds and 0.01 milliseconds, respectively. Should the evidence or the key be suppressed, their compromise is swiftly ascertained. The probabilistic detection hinges on the methodology employed to distinguish these two elements. Remarkably, the comparison itself concludes in a mere 0.001 milliseconds upon the discovery by the decryptor D.

We then assess the execution time of each basic cryptographic primitive used in IBE, and report them in Table 1.*right*. In this table, we use the following symbols to represent the running time of each function. T_{bp}: The

Table 1: Theoretical complexity (L) and **experimental time on IBE** (R): (a) Line 2 of Portex.EvidenceKeyManage in Figure 3; (b) Line 9-14; (c) Line 4-7; (d) Line 1-8 of GE on input $Manage$; (e) It costs 3.0121 ms for running KGen$_{PKG}$ in CPU.

		Algorithm	**Estimated Time** (ms)	**Experiment Time** (ms)	**Size**	**Environment**
Setup		HW.Load	-	342.9301	1.5MB	inside TEE
		MT.Init	-	<0.001	<1KB	outside TEE
		PKE.KGen	-	1.0413	<1KB	outside TEE
		S.KGen	-	1.8399	<1KB	outside TEE
Enc		Enc.Setup	-	0.3119	-	outside TEE
		IBE.Enc	$2T_{bp} + 2T_{ep.g1} + 2T_{ep.gt} + T_{bm} \approx 1.8535$	2.0010	3.4KB	outside TEE
EvidenceKeyManage	D	IBE.KGen$_{D1}$	$2T_{ep.g1} + T_{ep.gt} + T_{bm} \approx 1.2544$	1.3170	1.2KB	outside TEE
		Manage.SendReq[a]	$T_{sign} \approx 1.6972$	7.1164	-	outside TEE
		Manage.Verify[b]	$T_{verify} + T_{dec} \approx 7.5307$	17.9325	-	outside TEE
		IBE.KGen$_{D2}$	$4T_{bp} + 3T_{ep.g1} + T_{ep.gt} + 5T_{bm} \approx 2.9666$	3.1452	3.1KB	outside TEE
	LM	Manage.LogGen[c]	$T_{sign} \approx 1.6972$	6.9579	-	outside TEE
	PKG	Manage.LogVerify[d]	$T_{verify} \approx 6.0527$	10.6760	-	outside TEE
		IBE.KGen$_{PKG}$	$4T_{ep.g1} + 3T_{bm} + T_{hw} \approx 2.4252$	2.5676[e]	3.0KB	inside TEE
		Manage.SendPkey	$T_{sign} + T_{enc} \approx 3.4893$	16.6631	-	outside TEE
Dec		Dec.Setup	-	0.1128	-	outside TEE
		IBE.Dec	$2T_{bp} + T_{ep.gt} + 3T_{bm} \approx 0.6035$	0.7079	3.9KB	outside TEE
CheckViolation		CheckViolation	-	0.1599	-	outside TEE
Detection		Forged_evidence	-	1.0677	<1KB	outside TEE
		Forged_key	-	3.5497	<1KB	outside TEE
		Suppress_evidence	-	<0.001	<1KB	outside TEE
		Suppress_key	-	<0.001	<1KB	outside TEE
		LogInspect	$2 \cdot (4T_{bp} + T_{ep.gt} + 2T_{bm}) \approx 2.3076$	2.3586	3.7KB	outside TEE

Symbol	**Size** (B)
pk_{pke}	148
sk_{pke}	180
vk_{sign}	148
sk_{sign}	180

Symbol	**Time** (ms)
$T_{ep.g1}$	0.6028
$T_{ep.gt}$	0.0473
T_{bm}	0.0015
T_{hw}	0.0096

Symbol	**Time** (ms)
T_{enc}	1.7921
T_{dec}	1.4780
T_{sign}	1.6972
T_{verify}	6.0527

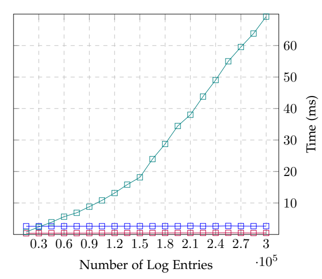
execution time of one bilinear pairing operation $e(P, Q)$, where $\{P, Q\} \in \mathbb{G}$; $T_{ep.g1}$: The execution time of one exponentiation operation P^x, where $P \in \mathbb{G}, x \in \mathbb{Z}_p$; $T_{ep.gt}$: Except for $P \in \mathbb{G}_T$, similar to $T_{ep.g1}$; T_{bm}: The execution time of one scale multiplication operation $P \cdot Q$, where $\{P, Q\} \in \mathbb{G}$; T_{hw}: The time of context switch to TEEs. These results closely align with our estimated times for each primitive, which offers robust evidence supporting the accuracy of our experimental findings.

Evaluation with dynamic parameters. To determine if deployment might be feasible in practice, we further evaluate how scalability is affected by dynamic parameters. We consider two scenarios: the performance overhead of decryption, evidence tracing, and compromise detection under (a) escalating sizes of security parameters and (b) an increasing number of decryption instances.

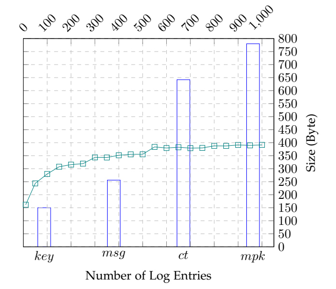
Intuitively, a larger security parameter enhances resilience against attacks. This enhancement comes at the cost of increased execution time for cryptographic operations. Empirical findings align with our expectations: *The time costs of encryption, evidence-key management, and decryption consistently increase with higher security parameter settings*. This phenomenon stems primarily from the augmented computational load of the underlying sub-algorithms. For example, the execution time of IBE.KGen$_{D_2}$ within the key-request algorithm expands from 4.2ms at a security parameter of 512 to 11.2ms at a heightened security parameter of 1024 (cf. Figure 5(c)). Nonetheless, the evidence tracing and compromise detection exhibit relatively steady performance, as they involve fewer time-intensive primitives.

We now consider the performance overhead arising from an escalating number of decryption operations (cf. Figure 5(a), 5(e) and 5(f)). To illustrate, we consider a specific scenario where the number of decryption operations grows from an initial $1,000$ to a substantial $300,000$. Our evaluation reveals that encryption time remains invariant with increased decryption operations as it is independent with TS, relying solely on the decryptor's ID and the encryptor's sequence number provided. In contrast, the time required for decryption directly correlates with the historical count of decryption operations. In particular, the execution time of operations such as evidence generation and evidence verification ($L.8.7$-$L.8.11$ in Figure 3), both integral to TS, experiences an increment. Consequently, this rise in processing time within TS contributes to an overall elongation of the decryption process. Meanwhile, the execution time of the trace algorithm is directly correlated with the size of the log. As the log size increases, the time required to query a malicious decryption operation from the log also increases. For example, when the log contains 21,000 entries, the tracing algorithm takes 38.5ms to finish the query. Upon augmenting the log size of 300,000 entries, the execution time of the trace algorithm increases to 69.2ms. Notably, the time cost of compromise detection is stable. Also, to maintain the integrity of evidence during decryption, it is required to choose a new sequence number for each encryption. As depicted in Figure 5, when the sequence number length is increased, the running time for evidence-key management displays a clear increasing trend. Fortunately, the impact on the running time for encryption and decryption is relatively minimal.

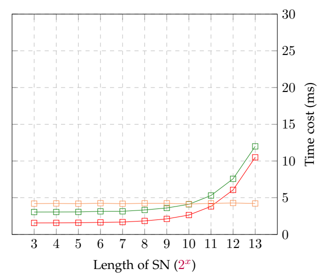
Performance comparison with other solutions. Table 2 compares



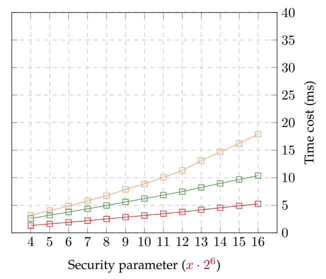
(a) Evidence verification (purple), generation (blue), and tracing (teal)



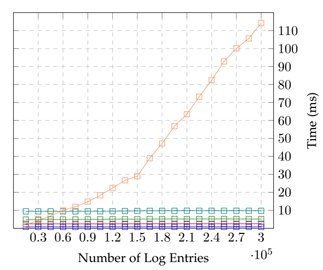
(b) The size of evidence (teal) and *key*, *ct*, *msg*, *mpk* (blue)



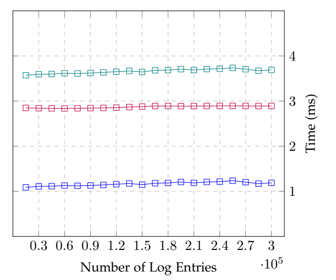
(c) KGen_{D1} (red), KGen_{PKG} (green), and KGen_{D2} (yellow) in algorithm IBE



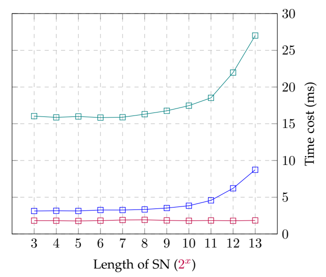
(d) KGen_{D1} (red), KGen_{PKG} (green), and KGen_{D2} (yellow) in algorithm IBE



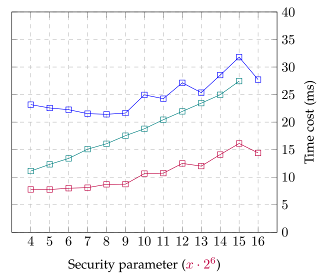
(e) Encryption (purple), Decryption (blue), Evidence-key management by D (teal), by LM (yellow), and by PKG (green)



(f) Deterministic detection of Forged evidence (purple), of Forged key (teal), and probabilistic detection (blue)



(g) Encryption (blue), Evidence-key management (teal), Decryption (purple)

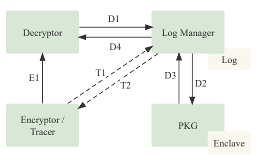


(h) Encryption (blue), Evidence-key management (teal), Decryption (purple)

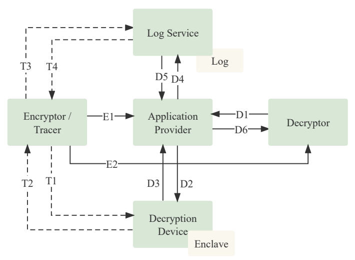
Figure 5: Experimental performance results

Table 2: Comparison between ours and related studies by analyzing the computational complexity of functional operations, the operations including Inter-party Interactions (T_{inter}), Encryption (T_{enc}), Decryption (T_{dec}), Signing (T_{sign}), Verification (T_{verify}).

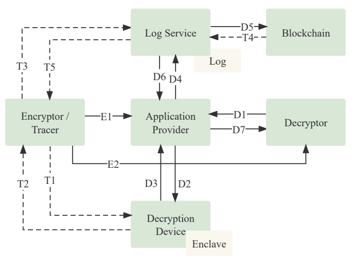
Solutions	**Encryption**	**Decryption**	**Tracing**
Ours $\mathbb{P}_o$	$T_{inter} + T_{enc} + T_{sign}$	$4T_{inter} + T_{enc} + 2T_{dec} + 3T_{sign} + 4T_{verify}$	$2T_{inter} + T_{sign} + 2T_{verify}$
Severinsen's [32]	$2T_{inter} + T_{enc} + T_{sign}$	$6T_{inter} + T_{dec} + T_{sign} + 2T_{verify}$	$4T_{inter} + T_{sign} + T_{verify}$
Luo's [42]	$2T_{inter} + T_{enc} + T_{sign}$	$7T_{inter} + T_{enc} + T_{dec} + T_{sign} + 2T_{verify}$	$5T_{inter} + T_{dec} + 2T_{sign} + 2T_{verify}$
Fialka [4]	$2T_{inter} + 2T_{enc} + 2T_{sign} + T_{verify}$	$N * T_{inter} + T_{dec} + T_{verify}$	$6T_{inter} + 2T_{dec} + T_{sign} + 2T_{verify}$



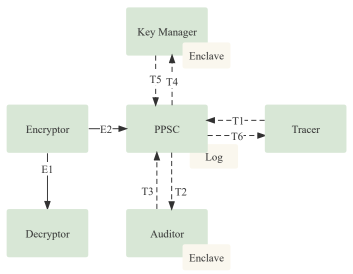
(a) Ours



(b) Severinsen's [32]



(c) Luo's [42]



(d) Fialka [4]

Figure 6: Comparing operational mechanism of Encryption-Decryption protocol ($\xrightarrow{E/D}$) and Tracing protocol ($\overset{T}{\dashrightarrow}$)

the performance of our proposed solution with that of existing works. As different works use different decryption algorithms, conducting a direct comparison based on actual runtime performance is unfair. Therefore, we examine the workflows of these protocols and quantify the frequency of critical operations within each protocol to analyze their computational costs. These critical operations contain inter-party interactions T_{inter}, encryption T_{enc}, decryption T_{dec}, signing T_{sign}, and signature verification T_{verify}. The inter-party interaction T_{inter} refers to the communication or exchange of information between different parties involved in a protocol, such as between the encryptor and potentially a third-party entity like a storage center or log manager.

Our solution has less interaction complexity compared to prior related works. For example, Severinsen's protocol ($\mathbb{P}_{sever}$), Luo's protocol ($\mathbb{P}_{luo}$), and Fialka ($\mathbb{P}_{fialka}$) mandate that the encryptor sends a copy of the ciphertext or auxiliary information to a storage center (termed as the Application Provider in $\mathbb{P}_{sever}$ and $\mathbb{P}_{luo}$, and Privacy-Preserving Smart Contracts(PPSC) in $\mathbb{P}_{fialka}$, cf. Figure 6) in addition to the intended recipient. In contrast, our protocol ($\mathbb{P}_o$) eliminates the need for encryptors to send an additional ciphertext backup, thereby obviating the requirement for a storage center and reducing interaction complexity. In $\mathbb{P}_o$, the encryption, decryption, and tracing operations require T_{inter}, $4T_{inter}$, and $2T_{inter}$, respectively.

Furthermore, we present the computational complexities associated with encryption and decryption in different protocols (see Table 2). The protocol $\mathbb{P}_{sever}$ requires one encryption and one decryption ($T_{enc} + T_{dec}$), while the protocols $\mathbb{P}_{luo}$, $\mathbb{P}_{fialka}$, and $\mathbb{P}_o$ each necessitate an additional encryption and decryption ($2T_{enc} + 2T_{dec}$). Although $\mathbb{P}_{sever}$ involves fewer encryption and decryption operations, it does not consider the message transmission after encryption, rendering the protocol incomplete.

Space evaluation. We also consider the feasibility of storage requirements. First, we evaluated the storage usage across four types of data (namely, key, ct, msg, and mpk). Our evaluations were conducted using SHA256 hashing, which generates a 32-byte length digest. Results indicate that the storage load for mpk with 1,000 entries requires approximately 780 bytes of storage, which is the most significant usage we observed. Secondly, we evaluated the trend of evidence size as the number of log entries increased. We found that the trend remains relatively stable, indicating the evidence size will occupy a constant amount of storage.

9 Discussion

We clarify the difference between the concepts of access control, traceability, and accountability, and provide potential applications to emphasize the necessity of our scheme.

9.1 Access Control, Traceability, and Accountability

The access control describes *who and when* can access data and resources. It prevents unauthorized access before it occurs. Traceability records the activities of how the data has been created, modified, accessed, or transferred. It starts to work after accessing the resources. Accountability holds individuals responsible for their actions (based on traceability), emphasizing the punishment of inappropriate access (see Table 3).

Table 3: Description and example

Concept	Description	Example
Access control	Governing who can access specific resources, defining user permissions, privileges, and restrictions.	A nurse may access patient records in his/her department but not in other departments.
Traceability	Providing a documented trail of how the data has been created, modified, accessed, or transferred.	A nurse accesses a patient's record in his/her department. The system records the nurse's name, time of access, and the reason for access (e.g., treatment, diagnosis).
Accountability	Checking the documented trail to make users accountable for any misuse.	A nurse has improperly accessed a patient's records he/she is not allowed to access. The improper access to the data cannot be denied.

In the context of accountable decryption, the access control can be more fine-grained, e.g., one can even specify the legitimate decryption timeframe. For example, one can ask the decryptor to decrypt only between 3:00 PM and 5:00 PM on some day. Traceability underscores the recording of the specific decryption time to ensure a traceable trail. Meanwhile, accountability focuses on verifying whether decryption occurred within the specified time frame. If decryption takes place outside the permitted timeframe, the decryptor is accountable for its behavior and the owner of the data is aware of such behavior.

9.2 Potential Applications

Accountable ePHI. Electronic protected health information (ePHI) [43] refers to any health information that can identify a patient. Balancing ePHI privacy and accessibility is challenging. Sharing ePHI to all doctors increases the potential risk of information leakage. Conversely, sharing the needed health information with only authorized doctors and meanwhile making any improper access undeniable is crucial. While one can use conventional access control mechanisms to achieve the same goal, this can slow down treatment as additional access control procedures are required, which is not desirable during emergencies. Our system provides a solution to address the challenges. It allows for information sharing while auditing unauthorized access and potential breaches. We consider patients as encryptors (i.e., E) who possess sensitive data, such as congenital genetic disorders. E encrypts their data and stores it in the hospital's cloud server, making it inaccessible to general practitioners (e.g., dental consultants). For instance, when a patient requires emergency treatment, dentists (i.e., D) can bypass rules and access the ePHI. After that, a designated auditor J (e.g., regulatory authorities) accesses the audit trail to monitor whether patient data are being accessed for the right purpose.

Transparent KYC. Know Your Client (KYC) standards are employed in the investment industry to verify customer identities and assess their risk and financial profiles [44]. Customers are required to provide KYC information to their banks, enabling the latter to conduct anti-money laundering procedures. However, this practice raises concerns regarding the potential misuse of customer data. Our solution addresses these concerns by allowing the customers to upload encrypted data and become aware of who have accessed the data. This approach permits banks to access private information only when necessary.

Accountable warrant. In an ideal scenario, law enforcement agencies could access citizens' data stored in the cloud center based on electronic warrants [45]. However, in practice, warrants are processed confidentially to maintain the integrity of investigations and abuse by malicious law enforcement agencies are not in place [46]. Our solution addresses these concerns by requiring the law enforcement (D) to maintain a documented trail when accessing an individual (E)'s encrypted data. This trail enables public verification of law agencies' decryption activities and ensures the safe and legal use of warrants, thereby mitigating the risks of abuse.

10 Related work

Trusted hardware has been proposed as an effective tool for achieving accountability protocols. Pasture et al. [47] devised a secure messaging and logging library incorporating offline data access safeguarded by trusted hardware. Their approach ensures that access actions are irrefutable and revocations are verifiable. Ryan [3] proposed an accountable decryption protocol assisted by trusted hardware. Under the security of trusted hardware devices, the private key is stored inside TEEs, and the information of each decryption will be recorded in a transparent log. Severinsen [32] implemented Ryan's protocol using Intel SGX [13], guaranteeing that encryptors can detect every decryption. In this scheme, decryptors can only decrypt the ciphertext after updating the Merkle tree in logs. Liang et al. [48] proposed a decentralized accountable system for healthcare data using Intel SGX and blockchain. Luo et al. [42] applied the blockchain and SGX to an accountable decryption scheme. The decryptor is assisted by SGX, similar to Severinsen's solution, but the log information is recorded on the blockchain. Fialka [4] employed a TEE-based confidential smart contract to make decryption accountable, where the transaction is used as evidence to trace a user's decryption.

Different from prior studies that merely provide prototypes, our work formally defines the security properties of accountable decryption and proves the correctness of our concrete construction. Besides, this line of work often assumes that TEEs are fully trusted. If the hardware is compromised, it is not hard to see that the solutions no longer work. Worse still, overseers cannot learn whether the hardware has been compromised. To our knowledge, our scheme is the first work that considers compromised TEEs. This is achieved by introducing a detection algorithm as part of our solution.

11 Conclusion

In this paper, we focus on achieving the practical accountability of decryption operations. We introduce a novel set of definitions tailored for accountable decryption. Our definition aims to capture all possibilities and impossibilities in the formulation. We further construct a scheme aligning with the definitions by using trusted hardware. Our prototype and evaluations prove the practicability and efficiency.

References

[1] Giampaolo Bella and Lawrence C Paulson. Accountability protocols: Formalized and verified. *TISSEC*, 9(2):138–161, 2006.

[2] Joan Feigenbaum, James A Hendler, Aaron D Jaggard, Daniel J Weitzner, and Rebecca N Wright. Accountability and deterrence in online life. In *WEBSCI*, pages 1–7, 2011.

[3] Mark D Ryan. Making decryption accountable. In *Cambridge International Workshop on Security Protocols*, pages 93–98. Springer, 2017.

[4] Rujia Li, Qin Wang, Feng Liu, Qi Wang, and David Galindo. An accountable decryption system based on privacy-preserving smart contracts. In *ISC*, pages 372–390. Springer, 2020.

[5] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel Weitzner. Practical accountability of secret processes. In *USENIX Security*, pages 657–674, 2018.

[6] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *OSR*, 41(6):175–188, 2007.

[7] Daniel J Weitzner, Harold Abelson, Tim Berners-Lee, Joan Feigenbaum, James Hendler, and Gerald Jay Sussman. Information accountability. *CACM*, 51(6):82–87, 2008.

[8] Rajashekar Kailar. Accountability in electronic commerce protocols. *TSE*, 22(5):313–328, 1996.

[9] Aydan R Yumerefendi and Jeffrey S Chase. The role of accountability in dependable distributed systems. In *Proceedings of HotDep*, volume 5, pages 3–3. Unseix Association, 2005.

[10] Ruth W Grant and Robert O Keohane. Accountability and abuses of power in world politics. *APSR*, 99(1):29–43, 2005.

[11] Joan Feigenbaum, Aaron D Jaggard, and Rebecca N Wright. Towards a formal model of accountability. In *NSPW*, pages 45–56, 2011.

[12] B Lampson. Notes for a presentation entitled "accountability and freedom,. *http://research.microsoft.com/en-us/um/people/blampson/slides/AccountabilityAndFreedom.ppt*, 2005.

[13] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016(86):1–118, 2016.

[14] Sandro Pinto and Nuno Santos. Demystifying ARM Trustzone: A comprehensive survey. *CSUR*, 51(6):1–36, 2019.

[15] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *EuroSys*, pages 1–16, 2020.

[16] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. VoltPillager: Hardware-based fault injection attacks against Intel SGX enclaves using the SVID voltage scaling interface. In *USENIX Security*, pages 699–716, 2021.

[17] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *CCS*, pages 1741–1758, 2019.

[18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[19] Weili Wang, Sen Deng, Jianyu Niu, Michael K Reiter, and Yinqian Zhang. Engraft: enclave-guarded raft on byzantine faulty nodes. In *CCS*, pages 2841–2855, 2022.

[20] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. Memory corruption attacks within android tees: a case study based on op-tee. In *ARES*, pages 1–9, 2020.

[21] Craig Gentry. Certificate-based encryption and the certificate revocation problem. In *EUROCRYPT*, pages 272–293. Springer, 2003.

[22] Sattam S Al-Riyami and Kenneth G Paterson. Certificateless public key cryptography. In *ASIACRYPT*, pages 452–473. Springer, 2003.

[23] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. Iron: Functional encryption using Intel SGX. In *CCS*, pages 765–782, 2017.

[24] Intel SGX SDK. *https://software.intel.com/en-us/sgx-sdk*.

[25] Helen Nissenbaum. Accountability in a computerized society. *Science and Engineering Ethics*, 2:25–42, 1996.

[26] Ahto Buldas, Helger Lipmaa, and Berry Schoenmakers. Optimally efficient accountable time-stamping. In *PKCW*, pages 293–305. Springer, 2000.

[27] Aydan R Yumerefendi and Jeffrey S Chase. Strong accountability for network storage. *TOS*, 3(3):11–es, 2007.

[28] Andreas Haeberlen. A case for the accountable cloud. *OSR*, 44(2):52–57, 2010.

[29] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multiparty computation with identifiable abort. In *CRYPTO*, pages 369–386. Springer, 2014.

[30] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

[31] Andreas Kogler, Daniel Gruss, and Michael Schwarz. Minefield: A software-only protection for SGX enclaves against DVFS attacks. In *USENIX Security*, 2022.

[32] Kristoffer Myrseth Severinsen. Secure programming with intel SGX and novel applications. Master's thesis, 2017.

[33] Fred Cohen. The use of deception techniques: Honeypots and decoys. *Handbook of Information Security*, 3(1):646–655, 2006.

[34] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *NDSS*, 2020.

[35] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *CRYPTO*, pages 213–229. Springer, 2001.
[36] Benoît Libert and Damien Vergnaud. Towards black-box accountable authority IBE with short ciphertexts and private keys. In *PKCW*, pages 235–255. Springer, 2009.
[37] Ben Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.
[38] The pairing-based cryptography library. *https://crypto.stanford.edu/pbc/*.
[39] MerkleCpp library. *https://github.com/merklecpp/merklecpp*.
[40] Openssl: Cryptography and ssl/tls toolkit. *https://www.openssl.org/*.
[41] Intel SGX SSL. *https://github.com/intel/intel-sgx-ssl*.
[42] Yili Luo, Jia Fan, Chunhua Deng, Yixin Li, Yue Zheng, and Jianwei Ding. Accountable data sharing scheme based on blockchain and SGX. In *CyberC*, pages 9–16. IEEE, 2019.
[43] Health insurance portability and accountability act. *https://hipaa.yale.edu/security/break-glass-procedure-granting-emergency-access-critical-ephi-systems*, 2004.
[44] José Parra Moyano and Omri Ross. KYC optimization using distributed ledger technology. *Business & Information Systems Engineering*, 59:411–423, 2017.
[45] Joshua A Kroll, Edward W Felten, and Dan Boneh. Secure protocols for accountable warrant execution. *https://www.cs.princeton.edu/~felten/warrant-paper.pdf*, 2014.
[46] Matthew Green, Gabriel Kaptchuk, and Gijs Van Laer. Abuse resistant law enforcement access systems. In *EUROCRYPT*, pages 553–583. Springer, 2021.
[47] Ramakrishna Kotla, Thomas L. Rodeheffer, Indrajit Roy, Patrick Stuedi, and Benjamin Wester. Pasture: Secure offline data access using commodity trusted hardware. In *OSDI*, 2012.
[48] Xueping Liang, Sachin Shetty, Juan Zhao, Daniel Bowden, Danyi Li, and Jihong Liu. Towards decentralized accountability and self-sovereignty in healthcare systems. In *ICICS*, pages 387–398. Springer, 2017.
[49] Michael Szydlo. Merkle tree traversal in log space and time. In *EUROCRYPT*, pages 541–554. Springer, 2004.
[50] Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security*, pages 317–334, 2009.
[51] Mark D Ryan. Enhanced certificate transparency and end-to-end encrypted mail. *Cryptology ePrint Archive*, 2013.
[52] Vipul Goyal. Reducing trust in the PKG in identity based cryptosystems. In *CRYPTO*, pages 430–447. Springer, 2007.
[53] Vipul Goyal, Steve Lu, Amit Sahai, and Brent Waters. Black-box accountable authority identity-based encryption. In *CCS*, pages 427–436, 2008.
[54] Zhen Zhao, Jianchang Lai, Willy Susilo, Baocang Wang, Yupu Hu, and Fuchun Guo. Efficient construction for full black-box accountable authority identity-based encryption. *IEEE Access*, 7:25936–25947, 2019.
[55] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *TOIT*, 31(4):469–472, 1985.
[56] Niels Ferguson and Bruce Schneier. *Practical Cryptography*, volume 141. Wiley New York, 2003.

APPENDIX A. CRYPTOGRAPHIC PRIMITIVES

Definition 15 (Discrete Logarithm Problem)**.** *Let* $N = p \cdot q$ *and* g *be a primitive root for both* $\mathbb{Z}_p^\star$ *and* $\mathbb{Z}_q^\star$*, where* p *and* q *are randomly safe primes. Given* $y = g^x \bmod N$*, it is computationally intractable to derive* x*.*

Merkle tree. Merkle Tree [49] MT is an optimized form of a hash list, which includes three algorithms.

- $\mathsf{MT.Init}(\lambda)$: This algorithm initializes a specific instance of the Merkle tree with a system-level input λ. Assuming the current tree contains a series of leaf nodes denoted as $(x_1, ..., x_{n-1})$.
- $\mathsf{MT.Insert}(x_n)$: This algorithm inserts a new element into the log (also known as a leaf attached to the tree). It takes as input a new element as the leaf node that contains the metadata x_n. The algorithm generates and outputs the updated root hash h where $h = \mathsf{MT}(x_1, ..., x_n)$ and evidence π. The operation represents the process of node merging and updating. The evidence π contains an array of sub-tree hashes that are used for verification.
- $\mathsf{MT.Verify}(\pi, x_n)$: This algorithm verifies whether the specified element, e.g., x_n exists in the tree and whether Merkle tree MT is append-only. It takes the evidence π and the queried element x_n, and outputs *true* if the verification succeeds. Otherwise, it outputs *false*.

In an append-only Merkle tree [50][51], data items are stored at the leaves, and new trees or items are added in a left-to-right chronological order. This structure allows for the verification of two crucial properties: (a) certain data is *contained* within the tree, and (b) a tree is an *extension* of another tree. Notably, both proof generation and verification have logarithmic complexity, along with the number of data entries increasing.

Identity-based encryption. The identity-based encryption (IBE) scheme [35] consists of the algorithms as follows.

- $\mathsf{IBE.Setup}(\lambda)$: The algorithm takes as input the security parameter λ, and outputs a master public key mpk and a master secret key msk.
- $\mathsf{IBE.KGen}(mpk, ID)$: The algorithm takes as input a user identifier ID, the master public key mpk and outputs a user's private key key.
- $\mathsf{IBE.Enc}(mpk, ID, m)$: This algorithm takes as input the master public key mpk, ID, a message m, and outputs a ciphertext ct.
- $\mathsf{IBE.Dec}(mpk, sk, ct)$: This algorithm takes as input mpk, sk, ct and outputs a message m.

Definition 16 (Correctness of IBE)**.** *An IBE scheme* IBE *achieves the correctness if for all* $m \in \mathcal{M}$ *and all pairs* $(ID, key) \leftarrow \mathsf{IBE.KGen}(\cdot)$*, it holds that*

$$\mathsf{IBE.Dec}(mpk, key, (\mathsf{IBE.Enc}(mpk, ID, m))) = m.$$

Definition 17 (IND-CCA Security of IBE)**.** *Consider an adversary* $\mathcal{A}$ *and a challenger* $\mathcal{C}$ *playing the following game.*

```
$\mathsf{G}_{\textit{IND-CCA}}(\lambda)$
1 : $\mathcal{C}$ runs $\mathsf{IBE.Setup}(\lambda)$ and $key \leftarrow \mathsf{IBE.KGen}$ with $ID$
2 : $\mathcal{C}$ sends $ID$ to $\mathcal{A}$;
3 : $\mathcal{A}$ adaptively chooses $ct$ and get back $\mathsf{IBE.Dec}(mpk, key, ct)$;
4 : $\mathcal{A}$ chooses two message $(m_0, m_1)$ and sends them to $\mathcal{C}$;
5 : $\mathcal{C}$ runs $ct^\star \leftarrow \mathsf{IBE.Enc}(mpk, ID, m_b)$, where $b \leftarrow\$ \{0,1\}$
6 : $\mathcal{C}$ sends $ct^\star$ to $\mathcal{A}$;
7 : $\mathcal{A}$ outputs its guess $b'$;
```

The advantage $\mathsf{Adv}_{\mathcal{A},\mathsf{IBE}}^{\mathsf{G}_{\textit{IND-CCA}}}(\lambda)$ *of* $\mathcal{A}$ *winning the game is*

$$\mathsf{Adv}_{\mathcal{A},\mathsf{IBE}}^{\mathsf{G}_{\textit{IND-CCA}}}(\lambda) = \Pr[b' = b] - \frac{1}{2}$$

IBE *achieves the IND-CCA security, if for all p.p.t. adversaries* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that* $\mathsf{Adv}_{\mathcal{A},\mathsf{IBE}}^{\mathsf{G}_{\textit{IND-CCA}}}(\lambda) < \mathsf{negl}(\lambda)$*. The details refer to [35].*

Extension on accountable authority IBE. Accountable authority identity-based encryption (A-IBE) is another technology route to achieve accountability. A-IBE was initially proposed by Goyal [52] as a means of reducing reliance on the trustworthiness of the private key generator (PKG) and making their behavior more accountable. Goyal's first full black-box A-IBE was proposed the following year [53], and subsequent research focused on improving its performance. For example, Libert et al. proposed an A-IBE scheme with shorter ciphertext and private key [36], and Zhao et al. reduced the computation cost of A-IBE [54]. These schemes provide solutions to trace whether a decoder box comes from the PKG or the user. Our solution was inspired by these ideas and employed a similar tracing mechanism to find the potential key leakage of the compromised Trusted Execution Environment (TEE).

Signature scheme. The signature scheme S [55] consists of three algorithms, defined as follows. We require the conventional unforgeability property for signatures.

- $\mathsf{S.KGen}(\lambda)$: This algorithm takes as input the security parameter λ, and outputs a pair of keys (vk, sk).
- $\mathsf{S.Sign}(sk, m)$: This algorithm takes as input a signing key sk and a message m, and outputs a signature σ.
- $\mathsf{S.Verify}(vk, \sigma, m)$: This algorithm takes as input the verification key vk, the signature σ, and the message m, and outputs *true* or *false*.

Definition 18 (EUF-CMA Security of S)**.** *Consider an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ playing the following game.*

$\mathsf{G}_{\textit{EUF-CMA}}(\lambda)$

```
1 : $\mathcal{C}$ runs $\mathsf{S.KGen}(\lambda)$ to generate keys $(vk, sk)$, and sends $vk$ to $\mathcal{A}$;
2 : Initializes a query set $\mathbb{Q} \leftarrow \{\}$;
3 : $\mathcal{A}$ chooses $m$ and get back $\mathsf{S.Sign}(sk, m)$ from $\mathcal{C}$, then $\mathbb{Q} \leftarrow \mathbb{Q} \cup m$;
4 : $\mathcal{A}$ forges and outputs a pair of message and signature $(m^\star, \sigma^\star)$;
```

The advantage $\mathsf{Adv}_{\mathcal{A},\mathsf{S}}^{\mathsf{G}_{\textit{EUF-CMA}}}(\lambda)$ *of $\mathcal{A}$ winning the game is*

$$\mathsf{Adv}_{\mathcal{A},\mathsf{S}}^{\mathsf{G}_{\textit{EUF-CMA}}}(\lambda) = \Pr[\mathsf{S.Verify}(\mathsf{vk}, \sigma^\star, \mathsf{m}^\star) = 1 | \mathsf{m}^\star \not\subset \mathbb{Q}]$$

S *achieves the property of EUF-CMA, if for all p.p.t. adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that* $\mathsf{Adv}_{\mathcal{A},\mathsf{S}}^{\mathsf{G}_{\textit{EUF-CMA}}}(\lambda) < \mathsf{negl}(\lambda)$.

Public key encryption. The PKE scheme [56] consists of three algorithms, defined as follows.

- $\mathsf{PKE.KGen}(\lambda)$: This algorithm takes as input the security parameter λ, and outputs a pair of keys (pk, sk). pk is the public key and sk is a secret key.
- $\mathsf{PKE.Enc}(pk, m)$: This algorithm takes as input a public key pk and a message m, and outputs a ciphertext ct.
- $\mathsf{PKE.Dec}(sk, ct)$: This algorithm takes as input a secret key sk and ciphertext ct, and outputs the message m.

Definition 19 (IND-CCA2 security of PKE)**.** *Consider an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ playing the following game.*

$\mathsf{G}_{\textit{IND-CCA2}}(\lambda)$

```
1 : $\mathcal{C}$ runs $(pk, sk) \leftarrow \mathsf{PKE.KGen}(\lambda)$, sends pk to $\mathcal{A}$;
2 : $\mathcal{A}$ adaptively chooses ct and get back $\mathsf{PKE.Dec}(sk, ct)$;
3 : $\mathcal{A}$ chooses two message $(m_0, m_1)$ and sends them to $\mathcal{C}$;
4 : $\mathcal{C}$ runs $ct^\star \leftarrow \mathsf{PKE.Enc}(pk, m_b)$, where $b \leftarrow\!\$ \{0,1\}$
5 : $\mathcal{C}$ sends $ct^\star$ to $\mathcal{A}$;
6 : $\mathcal{A}$ provides adaptively chosen $ct \neq ct^\star$,
    get back $\mathsf{PKE.Dec}(sk, ct)$;
7 : $\mathcal{A}$ outputs its guess $b'$;
```

The advantage $\mathsf{Adv}_{\mathcal{A},\mathsf{PKE}}^{\mathsf{G}_{\textit{IND-CCA2}}}(\lambda)$ *of $\mathcal{A}$ winning the game is*

$$\mathsf{Adv}_{\mathcal{A},\mathsf{PKE}}^{\mathsf{G}_{\textit{IND-CCA2}}}(\lambda) = \Pr[b' = b] - \frac{1}{2}$$

PKE *achieves the property of IND-CCA2, if for all p.p.t. adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that* $\mathsf{Adv}_{\mathcal{A},\mathsf{PKE}}^{\mathsf{G}_{\textit{IND-CCA2}}}(\lambda) < \mathsf{negl}(\lambda)$.

Definition 20 (Collision resistance of hash function)**.** *Consider a hash function $\Pi = (\mathsf{Gen}, \mathsf{Hash})$ and an adversary $\mathcal{A}$ playing the following game.*

$\mathsf{G}_{\textit{Hash-coll}}(\lambda)$

```
1 : Generate a key with $s \leftarrow \mathsf{Gen}(1^\lambda)$;
2 : $\mathcal{A}$ is given $s$ and outputs $\{x, x'\}$;
3 : Outputs 1 if and only if $(x \neq x') \cup (\mathsf{Hash}^s(x) = \mathsf{Hash}^s(x'))$;
    Otherwise, it outputs 0.
```

The advantage of $\mathcal{A}$ winning the game is $\mathsf{Adv}_{\mathcal{A},\Pi}^{\mathsf{G}_{\textit{Hash-coll}}}(\lambda) = \Pr[\mathsf{G}_{\textit{Hash-coll}}(\lambda) = 1]$. Π *achieves the property of collision resistance, if for all p.p.t. adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that* $\mathsf{Adv}_{\mathcal{A},\Pi}^{\mathsf{G}_{\textit{Hash-coll}}}(\lambda) < \mathsf{negl}(\lambda)$.

Appendix B. Detailed Implementation

Generation enclave (GE). Secrets in GE are the master secret key, the signing key, and the user's decryption keys (as in Listing 1). These secrets are initiated in the interface enclave_init. Specifically, this interface calls IbeAlgo.init() to set msk by giving a hardware-based random number. Then, it calls sgx_calculate_ecdsa_priv_key and sgx_ecc256_calculate_pub_from_priv to generate key pair vk_sign, sk_sign. Next, it creates an empty root hash pointer rth. The interface enclave_kreq is used to generate an incomplete Key cert for a user. It verifies the correctness of insertion request ir, POP and POE. If any of them is not SGX_Success, the enclave terminates the process. Otherwise, it returns pkey. The key generation is encapsulated into a class IbeAlgo.

```
element_t msk; // master secret key
sgx_ec256_private_t sk_sign; // signing key
uint8_t *rth; // root hash
```

Listing 1: The Secret of Generation Enclave.

Running IBE in TEEs. We implement IBE protocol as IbeAlgo. It mainly consists of three types of structures: master public key mpk_t, decryption key dk_t, and ciphertext ct_t. The master public key mpk is stored in the structure mpk_t, which contains $(\mathsf{X}, \mathsf{Y}, \mathsf{Z}[\mathsf{N}+1], \mathsf{h})$. We assume that CLIENT and TRACER have already obtained the mpk when setup. The dk_t contains $(\mathsf{d1}, \mathsf{d2}, \mathsf{d3})$, which are the components of a decryption key. Since the structure of incomplete key pkey is the same as the decryption key, dk_t can also be used to store the incomplete key.

The members of the class IbeAlgo include the bilinear pairing information pairing, the sizes of elements and structures size_comp_G1, size_Zr, ..., elements in algorithms Hz, theta, ..., and the algorithm functions of protocol IBE. Each element has a group type information, and there are four groups in a bilinear pairing: the input groups G1, G2, the output group GT, and the integer group Zr. The elements are initiated in the function IbeAlgo.init.

Log manager. We implement the log manager using a modified *Merklecpp* library, which contains the basic operations of the Merkle tree and supports generating Merkle proofs and verification. The log tree is in the type TreeT, where the hash size is 32 and the hash function is sha256_openssl. For each request from CLIENT with ID and SN, LM gets the system time ts. The structure Node is to store the log node information. After calculating the hash value hash of a Node type variable node, the hash will be inserted into the tree by logTree.insert(). In our implementation, the log tree appends one leaf and generates one ir for each request, and the POE can be simplified as the POP before the insertion.

TEE Inspect. We further present the TEE inspect algorithm.

```
typedef struct mpk_t {
    element_t X, Y, h, Z[N + 1];} mpk_t;
typedef struct dk_t {
    element_t d1, d2, d3;} dk_t;
typedef struct ct_t {
    element_t c1, c2, c3, c4;} ct_t;
class IbeAlgo {
public:
    pairing_t pairing
    int size_comp_G1, size_comp_G2, size_Zr,
        size_GT, ...;
    // a part of variables
    element_t Hz, t0, theta, ...;
    // a part of functions
    void init();
    void kgen1(int id);
    ...
}
```

Listing 2: Code of ibe.h Library.