

# Accountable Decryption made Formal and Practical

Rujia Li\*, Yuanzhao Li<sup>†</sup>, Qin Wang<sup>‡</sup>, Sisi Duan\*, Qi Wang<sup>†</sup>, Mark Ryan<sup>§</sup>

\*Tsinghua University, China

<sup>†</sup>Southern University of Science and Technology, China

<sup>‡</sup>University of New South Wales, Australia

<sup>§</sup>University of Birmingham, United Kingdom

**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 does not behave correctly, with non-negligible probability, his behavior will be detected, making the decryptor *accountable* for 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 hardware-assisted solution aligning with definitions. Instead of fully trusting the TEE like previous TEE-based accountability solutions, we take a further step, making TEE work in the “trust, but verify” model where a compromised state is detectable. Thirdly, we implement a full-fledged system and conduct 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

## I. 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][2], 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 [3].

In this work, we formalize the concept of *accountable decryption* [4][5], 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 infor-

mation 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 violation is detected, a judge imposes punishments on the decryptor. In this way, the decryptor is accountable for his behaviors.

Bringing accountability for decryption offers many benefits. For example, in electronic surveillance [6], the law-enforcement agency seeks access to users’ sensitive data from tech companies. Yet, some 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 in many applications [7]. We summarize these benefits as follows.

- *Detection of unauthorized access.* Accountable decryption allows an encryptor to audit decryption process [4].
- *Deterrents of illegal behaviour [8].* The decryptor is responsible for his actions, as any violation of the access control to the data can be caught and punished [9].
- *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.* Assuming a secure decryption scheme, where only users with private keys can decrypt ciphertexts. If decryptors discover that the received ci-

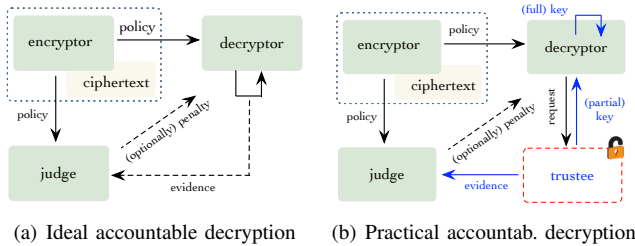
phertexts are decrypted without permission, users will be alerted of a potential leakage of decryption keys.

**Reviewing accountability definitions in the literature.** We review the development of the definitions in accountability and highlight the most relevant ones of our work. The research on accountability has a long-standing history. Its concept has been defined in various ways across different systems. Some key definitions are as follows.

- Nissenbaum (1996) [10] first introduced the term *accountability* in computer systems and mentioned accountability brings systems reliability and trustworthiness. The notion was followed by several works [11][12] that clarify the necessity of a combination of identity authentication and auditing to cover different scenarios.
- Subsequent studies (2000-.) emphasized honest behaviors. Indeed, accountability for honest behaviors is useful as non-tampering evidence and can be provided [2][8]. Regarding misbehaviors, these studies [13][14] focus on defining and modeling real-time detection mechanisms.
- A parallel line of works (2005-.) argued that the definition should encompass *after-the-fact* elements, incorporating a range of policy violations [9][15] and blame for misconduct [16][17] to enhance the system stability.

We extend existing accountability definitions to the scenario of accountable decryption. We delve into the theoretical underpinnings, culminating in a series of new definitions meticulously designed to meet the stringent security requirements inherent to this notion.

**Formalizing definitions for accountability of 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 6). First, our definitions ensure that dishonest decryptors can always be detected and punished (*non-repudiation*). Secondly, 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 reactions with appropriate punishments (as claimed in Section IV-A).



**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 [5]. 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 raises another concern for our definition, as the TS may become corrupted or untrustworthy. We thus propose new requirements for TS and our new definition (cf. Definition 7) emphasizes that TS should ensure the security of the managed key, and the *authenticity* and *completeness* of the decryption evidence.

**Building a practical accountable decryption scheme.** We construct a new scheme that achieves accountable decryption, called PORTEX. Our solution leverages trusted hardware as the trustee, specifically Trusted Execution Environments (TEEs) [18][19][20]. 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 (Definition 7.iii), making decryption accountable.

As our definition also captures the fact that the trustee may be corrupted, we introduced two algorithms to cope with TEE failures [21][22]. The first algorithm allows the TEE to store a partial key while the user securely holds another portion. Even if attackers gain access to the TEE and acquire some keys, they remain unable to deduce the complete decryption key. The second algorithm is based on deception technology [23]: it uses tactics to deceive malicious TEE into attacking the wrong targets and producing potentially useful information, thus detecting the potentially compromised state of TEEs. By these algorithms, our system is in accordance with Definition 8.vi-vii.

Notably, instead of fully trusting TEE like previous TEE-based accountability solutions [4][5][24][25], we make TEE work in the “trust, but verify” model where a compromised state is detectable. To our knowledge, PORTEX is the first *practical* solution to achieve accountability using TEE but NOT assuming TEEs are fully trusted. Indeed, by making TEEs traceable for their misbehaviors, our solution allows users (i.e., encryptors) to minimize their trust assumption.

**Providing full-fledged implementation and evaluations.** We provide a fully functional implementation<sup>12</sup> 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 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.

<sup>1</sup>Released at <https://github.com/portex-tee/portex>

<sup>2</sup>public service at <http://121.41.111.120>

## II. PRELIMINARIES

In this section, we provide the building blocks, including trusted hardware and cryptographic primitives.

### A. Trusted Hardware

We use the trusted execution environment (TEE) in this work. TEE is a secure area within the main processor that operates as an isolated kernel, ensuring the confidentiality and integrity of sensitive data and computations. State-of-the-art implementations of TEEs include Intel Software Guard Extensions (SGX) [18], ARM TrustZone [19], RISC-V Keystone [20], AMD SEV [26]. A TEE typically provides three main features: *runtime isolation*, *local/remote attestation*, and *sealing technologies*. 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 the trusted hardware. Sealing encrypts enclave secrets and provides persistent protection for the enclave secrets. By following some notions from [27], we define TEE as general trusted hardware HW as follows. We further provide a detailed construction in Appendix A.

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

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

**Notes.** In several TEE products (e.g., Intel SGX), to preserve signers’ anonymity, a quote is created using an anonymous group signature [18]. As anonymity is orthogonal to our formalization, we omit it in this work.

### B. Cryptographic Primitives

Our construction relies on standard cryptographic primitives. Due to space limitations, we provide two primitives while others are stated in Appendix B.

*Public key encryption.* A public key encryption scheme PKE consists of a tuple of PPT algorithms (PKE.Gen, PKE.Enc, PKE.Dec). PKE.Gen takes as input a secure parameter  $\lambda$  and outputs a public-private key pair  $(pk, sk)$ , written as  $(pk, sk) \leftarrow \text{PKE.Gen}(1^\lambda)$ . PKE.Enc takes as input a public key  $pk$  and a message  $m$  and outputs a ciphertext  $ct$ , denoted as  $ct \leftarrow \text{PKE.Enc}(pk, m)$ . PKE.Dec takes as input  $sk$  and  $ct$  and outputs a message  $m$ , written as  $m \leftarrow \text{PKE.Dec}(sk, ct)$ .

*Signature scheme.* A signature scheme  $S$  includes three PPT algorithms ( $S.Gen, S.Sign, S.Verify$ ).  $S.Gen$  takes as input  $\lambda$  and outputs a key pair  $(vk, sk)$ , written as  $(vk, sk) \leftarrow \text{PKE.Gen}(1^\lambda)$ .  $S.Sign$  takes as input a private key  $sk$  and a message  $m$  and outputs a signature  $\sigma$ , written as  $\sigma \leftarrow S.Sign(sk, m)$ . The deterministic algorithm  $S.Verify$  takes as input a verification key  $vk$ , a message  $m$ , and a signature  $\sigma$ . It outputs 1 if the signature is valid and 0 otherwise, written as  $b \leftarrow S.Verify(vk, \sigma, m)$ .

## III. DEFINITIONS OF ACCOUNTABILITY

Accountability has been defined in different contexts. In this section, we review the development of the definitions and highlight the most relevant ones of our work.

🔗 Nissenbaum (1996) [10] proposed the notion of accountability in computer science. While the paper does not provide a clear definition of accountability, it strengthens that accountability is crucial for building reliable computer science. For instance, “*accountability is valued because of its consequences for social welfare...Accountability can therefore be a powerful tool for motivating better practices, and consequently more reliable and trustworthy systems...*”

🔗 Kailar (1996) [11] and Yumerefendi et al. (2005) [12] extended the definition of accountability by emphasizing its objectivity and establishing a 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 2** (On linkage, merged by [11][12]). *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) [2] and Haeberlen et al. (2007) [8] embrace accountability as the process of furnishing evidence to a principal, which can subsequently be presented to the judge. Buldas et al. (2000) [13] and Yumerefendi et al. (2007) [14] delved into its application and strengthened the importance of real-time detection of misconduct.

**Definition 3** (On detection, rephrased). *Accountability in a computing system has the following features: (a) reli-*

able *evidence*, merged by [2][8]: 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 [13][14]: 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) [17], and Feigenbaum et al. (2011) [16], further improved the accountability definition by emphasizing that accountability is actually a posteriori behavior. These works emphasized the significance of consequences and deterrent measures in shaping responsible behavior in various domains. Grant et al. (2005) [15] and Weitzner et al. (2008) [9] underscored the imperative of validating adherence to predefined policies. The validation assesses whether a player strays from the predefined behaviors. If so, punitive measures can be taken.

**Definition 4** (On punishment, rephrased). *Accountability in a computing system implies the following properties: (a) awareness of policy violation, merged by [9][15]: 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 [15][16]: 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 works 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 works, such as Ishai et al. (2014) [29], have delved into establishing formal treatments of accountability.

**Definition 5** (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. Specifically, the system holds the following features: (1) linkage identities: each action is undeniably linked to the node 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.*

#### IV. ACCOUNTABLE DECRYPTION

In this section, we formally define the practical accountability of decryption.

##### A. General Principle of Accountability

We summarize the general principle of accountability in a computer system. It involves collecting automated evidence, detecting misbehavior, identifying policy violations, and attributing blame to the responsible participant.

- **Policy.** Policies define the expected behaviors of users within the system.
- **Action.** Users work on the operation that is expected to be aligned with the established policies.
- **Feedback.** Feedback on users' actions based on collected evidence allows for the assessment of compliance with policies in terms of users' actions.
- **Reaction.** Appropriate reactions, such as rewards or penalties, encourage accountability and deter non-compliant behavior.

However, even though considerable research has focused on achieving accountability in various domains, formalizing accountable decryption has remained unresolved.

##### B. Entities and Syntax

Accountable decryption 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. Their responsibilities are defined as follows.

- **Encryption.**  $(ct, \mathcal{P}) \leftarrow \text{Enc}(m)$ : An encryptor E executes this algorithm to generate a ciphertext  $ct$ , and policies  $\mathcal{P}$ . Here,  $\mathcal{P}$  dictates what are legal actions. For example, a dentist can only access a patient's sensitive records on family medical history in the event of a patient facing a life-threatening crisis that needs emergency treatment.
- **Decryption.**  $(m, \pi) \xleftarrow{\tilde{e}} \text{Dec}(key, ct)$ : A decryptor D executes this algorithm under designated environment denoted as  $\tilde{e}$ . This algorithm takes as input a private  $key$  and a ciphertext  $ct$ , alongside the encapsulated  $\tilde{e}$  environment variables. The outcome comprises both the plaintext  $m$  and a piece of evidence  $\pi$ , serving as substantiation for validating the decryption process. Notably,  $\tilde{e}$  captures critical aspects of the event, encompassing precise timing, unfolding sequence, and the identities of the participating entities. Ideally,  $\pi$  faithfully reports  $\tilde{e}$ .
- **Check.**  $tag \leftarrow \text{Check}(\pi, ct, \mathcal{P})$ : 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  $\pi$ , the policy  $\mathcal{P}$  and outputs a result  $tag \in \{true, false\}$ . Here, *true* indicates the decryptor's action is aligned with policies.
- **Reaction.**  $\perp \leftarrow \text{React}(tag, \mathcal{P})$ : A judge J imposes penalties against the decryptor in case of non-compliance.

Our definition follows the general principle. It has the ability to *trace*, *identify*, and *punish* malicious decryptors.

In particular, if the decryptor D maliciously accesses the plaintext, the judge J provides feedback on D's actions by checking evidence  $\pi$ , and then imposes penalties upon the D for his misbehaved decryption.

### C. Accountable Decryption

We formalize the notion of *accountable decryption*. It aims to hold decryptors responsible for their actions: malicious decryptors can be identified and penalized, while honest ones will never suffer from false accusations.

**Definition 6** (Accountability of decryption, ADec). *A system achieves ADec if the following conditions hold:*

- (i) *non-repudiation: for any execution of  $\text{Dec}(key, ct)$  on  $\tilde{e} \not\prec \mathcal{P}$ , there exists a negligible function  $\text{negl}$  that makes  $\text{Check}(\pi, \mathcal{P})$  output true, namely,  $\Pr[\text{true} = \text{Check}(\pi, \mathcal{P})] \leq \text{negl}(\lambda)$ .*
- (ii) *non-frameability: for any execution of  $\text{Dec}(key, ct)$  on  $\tilde{e} \prec \mathcal{P}$ , there exists a negligible function  $\text{negl}$  that makes  $\text{Check}(\pi, \mathcal{P})$  output false, namely,  $\Pr[\text{false} = \text{Check}(\pi, \mathcal{P})] \leq \text{negl}(\lambda)$ .*

Condition-(i) specifies that a decryptor who engages in misbehavior will be unequivocally detected. This means that if decryption occurs when it deviates from established policies, i.e.,  $\tilde{e} \not\prec \mathcal{P}$ , the decryption process cannot deny the occurrence of misbehavior. Namely,  $\text{Check}(\pi, \mathcal{P})$  always outputs *false*, resulting in the identification and subsequent penalization of such unscrupulous actors. Condition-(ii) guarantees the protection of honest decryptors from false accusations. When a decryptor operates within the bounds of legality, i.e.,  $\tilde{e} \prec \mathcal{P}$ , he shall not face any penalization.

**Example 1.** *Back to our case of electronic surveillance in Section I, Condition-(i) ensures that the law-enforcement agency who abuses the granted warrant can be detected and penalized, while Condition-(ii) prevents the wrongful framing of law-enforcement agencies in situations where the warrant has not been abused.*

**The need for additional trustee (TS).** Definition 6 seems sufficient for defining the accountability of decryption. However, in practice, there still exists a gap. In particular, decryption typically occurs locally. If a decryptor withholds  $\pi$  or provides false  $\pi$  (i.e.,  $\pi$  cannot reflect  $\tilde{e}$ ) during the decryption process, misbehaviors will never be captured. To address this challenge, we need a new role, and we call it *trustee* (TS). A trustee is responsible for managing *key* and  $\pi$  and ensures that the decryptor is accountable for his behavior.

Unfortunately, introducing a new role raises another concern when defining accountable decryption: the trustee may become corrupted. Indeed, the ability to control *key* and  $\pi$  put the trustee in a position where it can easily commit irregularities. Accordingly, we present two definitions of accountable

decryption with the trustee, considering a secure TS and a compromised TS.

**Definition 7** (Accountability of decryption with trustworthy trustee, ADec-TS). *The system achieves ADec-TS if the following conditions hold:*

- (i-ii) *the same as those in Definition 6.*
- (iii) *key-privacy: for any execution of  $\text{Dec}(key, ct)$ , the probability for TS to leak the key is negligible.*
- (iv) *evidence-authenticity: for any execution of  $\text{Dec}(key, ct)$  with  $\tilde{e}$ , the probability for TS to output a forged  $\pi'$  is negligible, where  $\pi \neq \pi'$ , and  $\text{Check}(\pi, ct, \mathcal{P}) = \text{Check}(\pi', ct, \mathcal{P})$ .*
- (v) *evidence-completeness: for any execution of  $\text{Dec}(key, ct)$ , the probability for TS to fail to output evidence is negligible, namely  $\Pr[m, \perp = \text{Dec}(key, ct)] \leq \text{negl}(\lambda)$ , where  $\perp$  signifies the absence of evidence being produced.*

This definition provides new requirements for newly introduced TS. Condition-(iii) assures the robust safeguarding of the decryption key, preventing any leakage. Condition-(iv) ensures that TS cannot generate forged evidence. Finally, Condition-(v) specifies that TS should always output evidence for each execution of decryption.

**Example 2.** *In our case of electronic surveillance introduced in Section I, the law-enforcement agency cannot finish the task of accessing users' sensitive personal data, and an additional witness is required. Condition-(iii) ensures that the witness never leaks the user's key; Condition-(iv) and Condition-(v) are put in place to safeguard the authenticity and integrity of the evidence gathered throughout the investigation.*

We consider the worst-case scenario where TS cannot be fully trusted. We require that even if TS is compromised, our ADec system is still secure, where the compromised TS cannot learn D's private key for decrypting the ciphertext. Additionally, the system has a certain probability of detecting and learning the compromised state of TS.

**Definition 8** (Accountability of decryption with untrusted trustee, ADec-uTS). *The system achieves ADec-uTS if the following conditions hold:*

- (vi) *leakage-resistance: Even if TS is compromised, the probability for TS to obtain D's private key is negligible.*
- (vii) *compromise-awareness: If TS fails to meet Condition-(iv) or Condition-(v) as specified in Definition 7, it exposes itself to the risk of detection. Alternatively, in the event of TS engaging in misconduct, the probability of the victim user's (i.e., encryptor, decryptor, judge) being **unaware** of TS's misbehavior is negligible.*

**Notes.** Our main goal, for Definition 8.(vi), is to restrict the ways in which TS can misbehave and seek to find potentially compromised state of TS instead of capturing all the misbehaviour of TS. It is evident that if TS misbehaves without leaving any discernible clues, the probability of detecting such misconduct is exceedingly low.

<sup>3</sup>Here, the symbol  $\not\prec$  indicates that  $\tilde{e}$  fails to adhere to the established pre-defined policies. To illustrate, consider a scenario in which a dentist accesses a patient's confidential records without the presence of a genuine emergency.

**Example 3.** Again, in our case of electronic surveillance introduced in Section I, Condition-(vi) ensures that the witness cannot learn the knowledge of the users’ full key, even if the witness is fully compromised; Condition-(vii) requires that external entities have some probabilities to learn about the witness’s misbehavior.

## V. A SECURE CONSTRUCTION

The main idea behind our construction is to run TS inside a TEE [18][19][20] and to force the action of key distribution to provide a tamper-evident evidence. TEE ensures the security of keys while enabling traceability in the key distribution process, thereby indirectly bringing decryption accountability.

### A. Strawman Solution

Briefly speaking, our naive solution involves four steps. ① E encrypts a message and sends the ciphertext to D. Meanwhile, E provides the decryption policy that defines the decryption condition. ② To decrypt the ciphertext, D has to submit 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 can access the secret by decrypting the ciphertext. ④ Finally, J checks the evidence and imposes penalties against malicious decryption.

The above approach immediately achieves Definition 6 and Definition 7 when we assume that TEE is fully trusted. Under this assumption, the codes will be completely executed as loaded, ensuring the integrity of the execution. If there is no key request from the decryptor, no evidence is produced and thus satisfies Condition-(ii). Conversely, whenever a key request is made, it leaves behind indisputable evidence and thus satisfies Condition-(i). Meanwhile, the secret keys never leave TEE, ensuring the security of decryption and satisfying Condition-(iii); The integrity of TEE’s execution ensures that the evidence is outputted accurately and completely, adhering to Condition-(iv); The evidence is signature-based, that ensures the unforgeability, thus satisfying Condition-(v).

### B. Technical Challenges

Definition 8 further assumes that the trustee can be compromised. Indeed, this is aligned with the fact that TEE products suffer from architectural vulnerabilities [21], side-channel attacks [22], and fault attacks [30]. Therefore, we need to provide a solution that satisfies Definition 8. However, a few technical challenges still exist in building a fully-fledged solution, as discussed below.

**Challenge-1: How to protect the 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.

We provide an efficient solution to mitigate challenges. In particular, we introduce a novel 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 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 [23]: 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  $P_e$  and decryption inspector  $P_d$ . They emulate conventional encryptors and decryptors, respectively. If TS outputs inconsistent results,  $P_e$  and  $P_d$  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.

### C. 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  $\pi$  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 [4][5][24][25], 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 works 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 must send a key request with a commitment for his random number to TS. ③ TS generates decryption keys and updates the evidence  $\pi$  of key extraction. Specifically, LM first stores D’s key request in a log and transforms it as evidence  $\pi$ , 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.

Our solution generally follows the strawman approach mentioned in Section V-B. The concrete decryption scheme follows the design of identity-based encryption (IBE) [31][32]. 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 [33] to improve security and efficiency. Below, we describe the technical details.

- (1) *Key management.* An encryptor  $E$  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 8.vi).
- (2) *Evidence management.* We use an append-only Merkle tree based log  $\mathcal{LOG}$  to store the decryption evidence  $\pi$ . 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  $\rho$  guarantees that the specific evidence is stored within a tree; the proof of extension  $\varepsilon$  proves that such a tree is an append-only extension of  $\mathcal{LOG}$ . We give an example to explain how to verify the  $\rho$  and  $\varepsilon$ . As shown in Figure 2, it depicts the Merkle tree before and after the insertion of node  $N_7$ . The  $\varepsilon$  is valid only if the root hash  $H$  is the hash of  $\{H(6, 6), H(4, 5), H(0, 3)\}$ , and the  $\rho$  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.

Our design also considered the worst-case scenario that a compromised  $TS$  may manipulate evidence and suppress evidence (adhering to Definition 8.vii). We accordingly propose a *detection algorithm* to detect such misbehaviors. Central to this algorithm lies the deployment of *inspectors*. In a bid to mimic

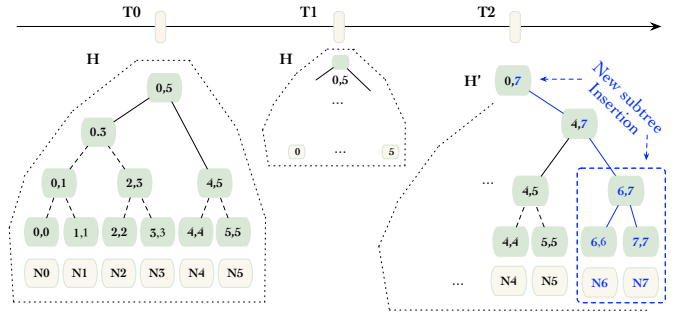


Figure 2: Merkle tree update

a conventional encryptor and decryptor,  $P_e$  and  $P_d$  dynamically engage in standard interactions with  $TS$  (behaving like a challenger). Based on the observations meticulously gathered by  $P_e$  and  $P_d$  throughout these interactions, our algorithm possesses the capability to effectively discern whether  $TS$  has been compromised. Drawing parallels, the detection function can be analogized to that of a loss prevention officer (LPO) in a shopping mall management case. This analogy aptly captures the mission—to thwart insider theft. Just as an LPO assumes the persona of a customer to detect any signs of employee theft,  $P_d$  embodies a similar role in our context. To elaborate, LPO selects some products, and if a salesperson fails to include all the products requiring settlement, then the LPO discerns the salesperson’s breach of trust. Back to our protocol, we discuss three sub-scenarios.

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

We then delve into the most tricky scenario:  $TS$  effectively conceals both the key and evidence. If  $TS$  refrains from any communication with  $P_d$ , its malicious actions may remain undetected. In our solution, the private key is generated based on  $P_d$ ’s commitment to an undisclosed random number, unbeknownst to  $TS$ . In the event that a decryptor,  $P_d$ , 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  $TS$ . Thus, for a given identity, the existence of private keys stemming from different sources acts as evidence against a deceitful  $TS$ . Admittedly, this relies on probabilistic grounds since  $P_d$  cannot guarantee a 100% probability of finding a malicious key. Nevertheless,

it at least exposes  $P_d$  to a risk of being detected.

- (3) TS, *privately generates key without leaving any evidence.* When  $P_d$  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,  $P_d$  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 [21][22]. 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.

## VI. FORMAL DEFINITIONS

We present the detailed protocol in Figure 3. We instantiate TEE using Intel SGX [18]. Notably, *L.x* represents line *x* in the pseudocode.

**Setup the system.**  $\text{Portex.Setup}(\lambda)$ : This is a preparation phase aiming to set up the system. The algorithm takes as input a security parameter  $\lambda$ , and outputs public parameters  $pm.s$ . In particular, it first calls  $\text{HW.Setup}$  to initiate the HW and generates  $pm.shw$  (L.1). Then, it runs  $\text{HW.Load}$  to load PKG code (e.g.,  $\text{Portex.KReq}$  as defined in Figure 3) into a generation enclave GE (L.2). Next, it runs (*“init”*,  $\lambda$ ) to set up the enclave and then publish the public keys ( $mpk, vk_{GE}$ ) (L.3). Meanwhile, the log manager LM calls  $\text{MT.Init}(\lambda)$  to initialize the log and runs  $\text{S.KGen}(\lambda)$  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 ( $pm.shw, mpk, vk_{LM/GE}, pk_{cli}^{enc}, vk_{cli}^{sign}$ ) are publicly accessible, and default inputs of the rest of algorithms.

① **The encryption phase.**  $\text{Portex.Enc}(ID, m)$ : 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  $\text{IBE.Enc}(mpk, ID|SN, m)$  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.

$\text{Portex.KReq}(ID, SN, \sigma_{cli})$ : The algorithm takes as input user identity  $ID$ , a serial number  $SN$ , and D’s signature  $\sigma_{cli}$ , and outputs a decryption key  $pkey$ , and the evidence  $\pi$  of key extraction. Inspired by [32], we separate the key generation into three sub-algorithms  $\{\text{IBE.KGen}_{D1}, \text{IBE.KGen}_{PKG}, \text{IBE.KGen}_{D2}\}$ . The detailed algorithms are shown as follows.

- (L.1-L.3): D first locally runs  $\text{IBE.KGen}_{D1}$  to select  $(t_0, \theta)$  and to generate a commitment  $C$ . Then, it gener-

ates a signature  $\sigma_{cli}$  on  $ID$  and  $SN$ . After that, it sends  $(ID, SN, C)$ , and  $\sigma_{cli}$  to LM.

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

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

③ **The trace phase.**  $\text{Portex.LogTrace}(\pi, ct, \mathcal{P})$ : In this phase, J examines the actions of the decryptor based on the evidence  $\pi$  stored in  $\mathcal{LOG}$ . Precisely, it finds  $\pi$  in  $\mathcal{LOG}$ , and then checks whether  $\pi$  satisfies  $\mathcal{P}$  (L.1-L.4). If no misbehavior is found, the protocol outputs *true*. Otherwise, it outputs *false*. To clarify this concept further, we provide three concise practical use cases.

- Consider a scenario where Alice can choose to share her location with some nominated friends and family members, referred to as “angels” (initiating  $\mathcal{P}$ ). The key point here is that only her designated angels have the privilege to access location information (leaving  $\pi$ ), and she (acting as J) retains the ability to monitor both the access and the timing of such access ( $\mathcal{LOG}$ ).
- Suppose Alice is being investigated by the police. In an effort to establish her innocence, she may choose to hand over her phone. With accountable decryption, Alice can upload her phone with the required contents ( $\mathcal{P}$ ). Then Alice (J) obtains evidence ( $\pi$ ) of what part of this uploaded material is decrypted ( $\mathcal{LOG}$ ).
- Assuming Alice provides know-your-customer (KYC) information to her bank, so that if necessary later, it can carry out anti-money laundering procedures ( $\mathcal{P}$ ). Recent proposals [24] suggest centralizing KYC registers



to make the procedure more efficient. With accountable decryption, the ill effects of such centralization (e.g.  $\pi \notin \mathcal{P}$ ) can be mitigated by making money laundering investigations (checking  $\mathcal{LOG}$ ) more transparent.

④ **The reaction phase.**  $\text{React}(tag, \mathcal{P})$ : The judge  $J$  imposes penalties against  $D$  who has conducted malicious decryption. These penalties encompass 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  $P_d$  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).

---

**Algorithm 1** The *detection algorithm*

---

```

compromised_state  $\in \{true, false\}$ 
1: -----deterministic detection-----
2: upon receiving  $\pi$  and  $pkey$  from  $TS$ 
3:    $key = \text{IBE.KGen}_{D_2}(mpk, ID, pkey)$ 
4:   if  $\pi \notin \mathcal{LOG} \wedge \text{Dec}(key, ct) \neq \perp$  then
5:     compromised_state = true  $\triangleright$  TS must have forged  $\pi$ 
6: upon receiving  $\pi$  and  $pkey$  from  $TS$ 
7:   if  $\pi \in \mathcal{LOG} \wedge \text{IBE.KGen}_{D_2}(mpk, ID, pkey) = \perp$  then
8:     compromised_state = true  $\triangleright$  TS must have forged  $pkey$ 
9: upon receiving  $pkey$  from  $TS$ 
10:   $key = \text{IBE.KGen}_{D_2}(mpk, ID, pkey)$ 
11:  if  $\text{find\_evidence}(\mathcal{LOG}) = \emptyset \wedge \text{Dec}(key, ct) \neq \perp$  then
12:    compromised_state = true  $\triangleright$  TS must have suppressed  $\pi$ 
13: upon receiving  $\pi$  from  $TS$ 
14:  if  $\text{find\_key}(\pi) = \emptyset$  then
15:    compromised_state = true  $\triangleright$  TS must have suppressed
     $pkey$ 
16: -----probabilistic detection-----
17: upon finding  $key'$ 
18:  if  $key \neq key' \wedge \text{Dec}(key', ct) \neq \perp$  then
19:    compromised_state = true
20:                                      $\triangleright$  TS must have impersonated  $D$ 

```

---

*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  $(t_0, \theta)$  are chosen randomly by  $P$  that are unknown to  $PKG$ ,  $PKG$  cannot generate a 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  $key$ , it demonstrates that  $D$  is maliciously created by  $TS$ , thus showing  $TS$ 's misbehavior.

## VII. SECURITY ANALYSIS

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

### A. Analysis Under Secure $TEE$

Assuming that  $TEE$  and cryptographic primitives are secure, our construction ensures that evidence cannot be suppressed or forged. We prove the theorem by contradiction. If  $\mathcal{A}$  made our construction fail to achieve properties, we could use such abilities to break our assumptions.

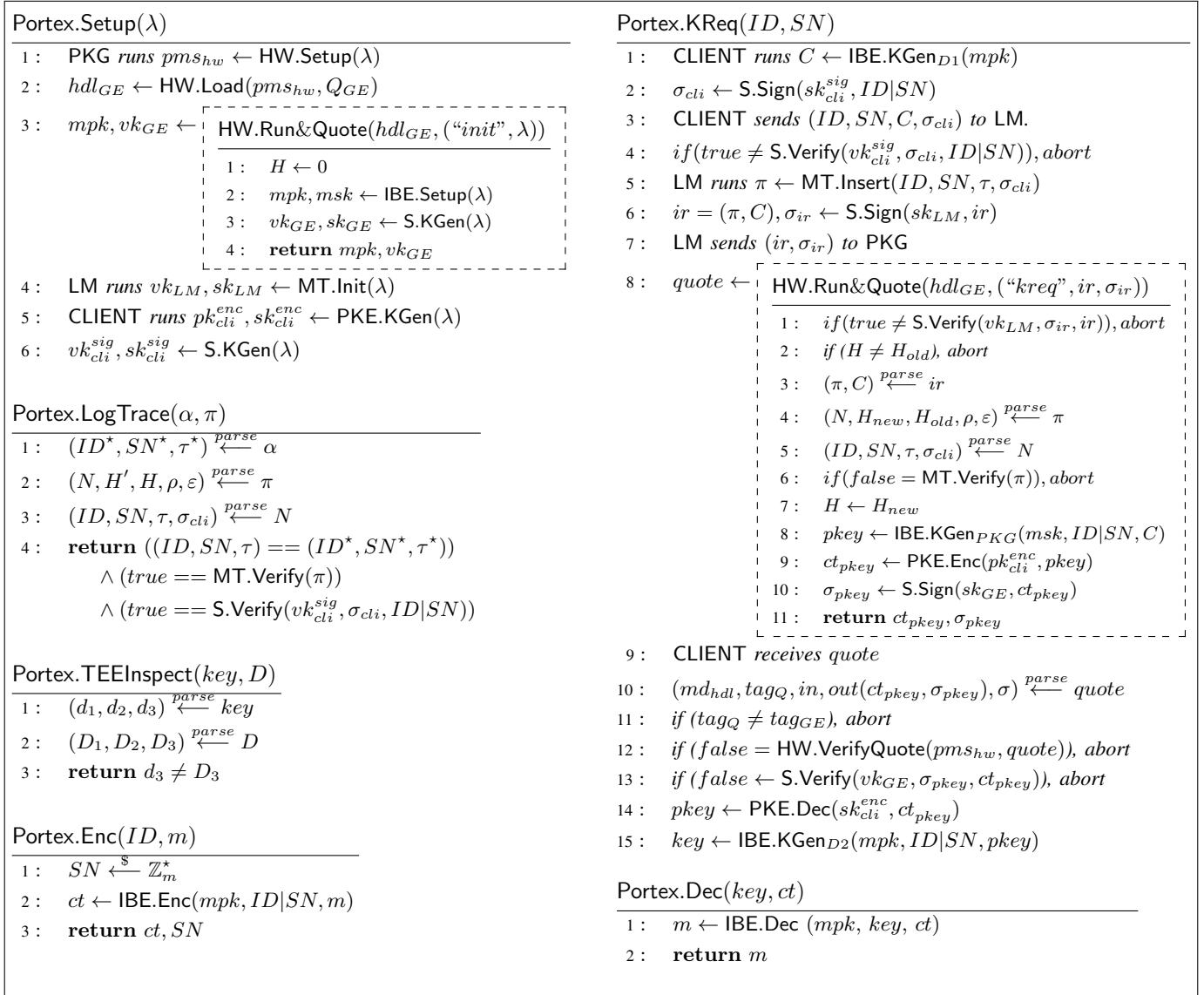
**Theorem 1.** *If  $HW$  is CodeUncy secure as defined in Definition 10 and  $S$  is EUF-CMA secure as defined in Definition 15, our construction achieves that (i) any  $D$  who engages in unauthorized decryption can be identified based on the evidence; and (v)  $TS$  cannot reject to output evidence.*

**Proof.** Suppose an adversary  $\mathcal{A}$  has obtained the decryption key but never leaves the decryption log. Then, we can use such abilities to break the execution integrity of  $TEE$  and the EUF-CMA security of  $S$ . As shown in Figure 3, when  $LM$  cannot furnish valid decryption evidence in the form of  $(ir, \sigma_{ir})$ , the algorithms L.8.1 and L.8.6 are programmed to abort. This further leads to the failure of the execution of algorithm L.8.8. Thus, if  $\mathcal{A}$  has successfully executed algorithm L.8.8, it implies that  $\mathcal{A}$  has either manipulated the  $TEE$  code (e.g.,  $\text{Portex.KReq}$  as defined in Figure 3) or provided the forged decryption evidence. However, the manipulation of the  $TEE$  code contradicts our assumption regarding the execution integrity of the  $TEE$ , as defined in Definition 10. Meanwhile, the forged evidence breaks the security of EUF-CMA of  $S$  as defined in Definition 15.  $\square$

**Theorem 2.** *If  $S$  is EUF-CMA secure as in Definition 15, our construction achieves that (ii) an adversarial  $D$  cannot frame an honest  $D$  by providing a forged  $\pi'$ ; (iv) an adversarial  $TS$  output a forged  $\pi'$ .*

**Proof.** Suppose an adversary  $\mathcal{A}$  has generated a log. According to the algorithm L.4, we have known that  $LM$  accepts the key request only if the signature  $\sigma_{cli}$  is valid, namely,  $S.\text{Verify}(vk_{cli}^{sig}, \sigma_{cli}, ID|SN) = true$ . The signing key  $sk_{cli}^{sig}$  is only known to  $\mathcal{A}$ . In this situation,  $\mathcal{A}$  generates a valid signature  $\sigma_{cli}^*$  without  $sk_{cli}^{sig}$ , which can be used to break the security of EUF-CMA of  $S$  as defined in Definition 15.  $\square$

**Theorem 3.** *If  $HW$  is StaConf secure as defined in Definition 11, the master private key  $msk$  used in our construction is securely protected.*



**Figure 3:** PORTEX protocol and generation enclave

**Proof.** If an adversary  $\mathcal{A}$  has obtained the decryption key, it implies that untrusted entities have accessed the sensitive state protected by TEE, which contradicts our assumption that TEE protects sensitive data and code from unauthorized access and unauthorized modification, as defined in Definition 11.  $\square$

### B. 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 under the standard assumption that cryptographic primitives are secure. Also, users can be aware of such compromises.

**Theorem 4.** *Our construction satisfies leakage resistance if the DLP assumption holds as defined in Definition 12.*

**Proof.** From the algorithm (L.2), we know that  $m = C_4 \cdot (\frac{e(C_1, d_1)}{e(C_2, d_2) \cdot C_3^{d_3}})^{-1}$ . where,  $d_1 = (Y \cdot h^{t_0 + t_1})^{1/x} \cdot H_Z(ID)^r$ ,  $d_2 = X^r$ ,  $d_3 = t_0 + t_1$ . As  $r, t_0, t_1$  is randomly selected from  $\mathbb{Z}_p^*$ ,

even if the adv fully controls  $x$ , it has a negligible probability of decrypting the ciphertext.

$$\begin{aligned}
m^* &= C_4 \cdot \left( \frac{e(C_1, d_1)}{e(C_2, d_2) \cdot C_3^{d_3}} \right)^{-1} \\
&= \frac{C_4 \cdot e(C_2, X^r) \cdot C_3^{t_1+t'}}{e(C_1, \frac{d_1'}{g^{\theta'}} \cdot H_Z(ID)^{r'})} \\
&= m \cdot \frac{e(g, Y)^s \cdot e(H_Z(ID)^s, X^r) \cdot e(g, h)^{s \cdot (t_1+t')}}{e(X^s, \frac{(Y R h^{t_1})^{1/x}}{g^{\theta'}} \cdot H_Z(ID)^r)} \\
&= m \cdot \frac{e(g, Y)^s \cdot e(g, h)^{s \cdot (t_1+t')} \cdot e(X^r, H_Z(ID)^s)}{e(g^{xs}, (\frac{Y R h^{t_1}}{g^{\theta'}})^{\frac{1}{x}}) \cdot e(X^s, H_Z(ID)^r)} \\
&= m \cdot \frac{e(g, Y)^s \cdot e(g, h^{t'} h^{t_1})^s}{e(g^s, \frac{Y h^{t_1} h^{t_0} \cdot X^\theta}{X^{\theta'}})} \\
&= m \cdot \frac{e(g, Y)^s \cdot e(g, h^{t_1})^s \cdot e(g, h^{t'})^s}{e(g^s, Y) \cdot e(g^s, h^{t_1}) \cdot e(g^s, h^{t_0})} \cdot e(g^s, X)^{(\theta' - \theta)} \\
&= m \cdot e(g^s, h^{(t' - t_0)} \cdot X^{(\theta' - \theta)})
\end{aligned} \tag{1}$$

When  $t' = t_0$  and  $\theta' = \theta$ , the output value is  $m^* = m$ . The adversary's sole source of information regarding  $t_0$  and  $\theta$  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 scenario can be equivalently reframed as a challenge in solving discrete logarithm problems, as outlined in Definition 12.  $\square$

**Theorem 5.** *Our construction satisfies compromise-awareness if the DLP assumption holds (Definition 12).*

**Proof.** We consider two possible 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/conceals 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.KReq, where  $d_1 = (Y \cdot h^{t_0+t_1})^{1/x} \cdot H_Z(ID)^r$ ,  $d_2 = X^r$ , and  $d_3 = t_0 + t_1$  for distinct  $t_0, t_1 \xleftarrow{\$} \mathbb{Z}_p^*$ . 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).

An adversary  $\mathcal{A}$  can compromise without detection only if it can manipulate the algorithm Portex.TEEInspect to output true when the  $d_3$  and  $D$  are equal. Consequently, the probability  $P^{\mathcal{A}}(\lambda)$  of the  $\mathcal{A}$ 's success is:

$$P^{\mathcal{A}}(\lambda) = \mathcal{P}[key.d_3 = D.d_3] \leq \frac{1}{p} \leq \frac{1}{2^\lambda} \tag{2}$$

Given these conditions, the adversary  $\mathcal{A}$ 's advantage is negligible, as it is limited by  $\frac{1}{2^\lambda}$ .  $\square$

## VIII. IMPLEMENTATION

We implement PORTEX in C++ and use Intel SGX SDK [34] as the TEE. The implementations of IBE algorithms, including key generation, encryption, decryption, and trace, are based on the Pairing-Based Cryptography (PBC) library [35].

Our Merkle Tree implementation is based on the merklecpp library [36]. The public key encryption and signature algorithms are based on the OpenSSL library [37]. SGX Enclave does not support the original OpenSSL, thus in the PKG program, we utilized SGXSSL instead [38]. We established a web server using the Drogon library [39] for the Log Manager program.

Additional details about the implementation have been provided in Appendix C. The source code is publicly accessible at <https://github.com/portex-tee/portex>, and the corresponding dependency libraries are listed below.

- Intel SGX SDK (v2.14): Developing SGX applications
- GMP (v6.2.1): The GNU Multiple Precision
- PBC (v0.5.14): Pairing-Based Cryptography
- Merklecpp (v1.0.0): A simple Merkle tree library
- OpenSSL (v1.1.1u): Cryptography and SSL/TLS Toolkit
- OpenSSL (v2.14): Intel Software Guard Extensions SSL
- Drogon (v1.8.3): HTTP application framework

## IX. EVALUATION

We evaluate the performance of PORTEX on a machine with 3.5GHz Intel Xeon CPU (Ice Lake). The symmetric bilinear pairing  $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$  is constructed on the curve  $y^2 = x^3 + x$ , and the security parameter  $\lambda$  is selected as 512. We report the average time of each algorithm by repeating the experiments 1,000 times.

### A. Performance and Scalability

This assessment 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.

*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. Also, we employ a security parameter of 512. Our evaluation shows that the full decryption process, including two sub-algorithms: *key request* and *decryption*, concludes within 10ms. In particular, key request operation takes approximately 1.31ms, 3.15ms, 2.56ms to finish IBE.KGen<sub>D1</sub>, IBE.KGen<sub>D2</sub>, and IBE.KGen<sub>PKG</sub>, 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

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

Algorithm		Estimated Time (ms)	Experiment Time (ms)	Size	Environment	Symbol	Size (B)	
Setup	HW.Load	-	342.9301	1.5MB	inside TEE			
	MT.Init	-	<0.001	<1KB	outside TEE	$pk_{pke}$	148	
	PKE.KGen	-	1.0413	<1KB	outside TEE	$sk_{pke}$	180	
	S.KGen	-	1.8399	<1KB	outside TEE	$vk_{sign}$	148	
Enc	Enc.Setup	-	0.3119	-	outside TEE	$sk_{sign}$	180	
	IBE.Enc	$2T_{bp} + 2T_{ep.g1} + 2T_{ep.gt} + T_{bm} \approx 1.8535$	2.0010	3.4KB	outside TEE			
KReq	D	IBE.KGen $_{D1}$	$2T_{ep.g1} + T_{ep.gt} + T_{bm} \approx 1.2544$	1.3170	1.2KB	outside TEE		
		KReq.SendReq <sup>a</sup>	$T_{sign} \approx 1.6972$	7.1164	-	outside TEE		
		KReq.Verify <sup>b</sup>	$T_{verify} + T_{dec} \approx 7.5307$	17.9325	-	outside TEE	Symbol	Time (ms)
		IBE.KGen $_{D2}$	$4T_{bp} + 3T_{ep.g1} + T_{ep.gt} + 5T_{bm} \approx 2.9666$	3.1452	3.1KB	outside TEE	$T_{ep.g1}$	0.6028
	LM	KReq.LogGen <sup>c</sup>	$T_{sign} \approx 1.6972$	6.9579	-	outside TEE	$T_{ep.gt}$	0.0473
		KReq.LogVerify <sup>d</sup>	$T_{verify} \approx 6.0527$	10.6760	-	outside TEE	$T_{bm}$	0.0015
PKG	IBE.KGen $_{PKG}$	$4T_{ep.g1} + 3T_{bm} + T_{hw} \approx 2.4252$	2.5676 <sup>e</sup>	3.0KB	inside TEE	$T_{hw}$	0.0096	
	KReq.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			
Trace	LogTrace	-	0.1599	-	outside TEE	Symbol	Time (ms)	
Detection	Forged_evidence	-	1.0677	<1KB	outside TEE	$T_{enc}$	1.7921	
	Forged_key	-	3.5497	<1KB	outside TEE	$T_{dec}$	1.4780	
	Suppress_evidence	-	<0.001	<1KB	outside TEE	$T_{sign}$	1.6972	
	Suppress_key	-	<0.001	<1KB	outside TEE	$T_{verify}$	6.0527	
	LogInspect	$2 \cdot (4T_{bp} + T_{ep.gt} + 2T_{bm}) \approx 2.3076$	2.3586	3.7KB	outside TEE			

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 I.right. In this table, we use the following symbols to represent the running time of each function.

- $T_{bp}$ : The 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 whether deployment can 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 (i) escalating sizes of security parameters and (ii) 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 cost of encryption, key request, and decryption consistently increase with higher security parameter settings.* This stems primarily from the augmented computational load of the underlying sub-algorithms. For example, the execution time of IBE.KGen $_{D2}$  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 (Figure 4(c)). Nonetheless, the evidence tracing and compromise detection exhibit relatively steady performance, as they involve fewer time-intensive primitives.

We consider the performance overhead arising from an escalating number of decryption operations (cf. Figure 4(a), 4(e) and 4(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

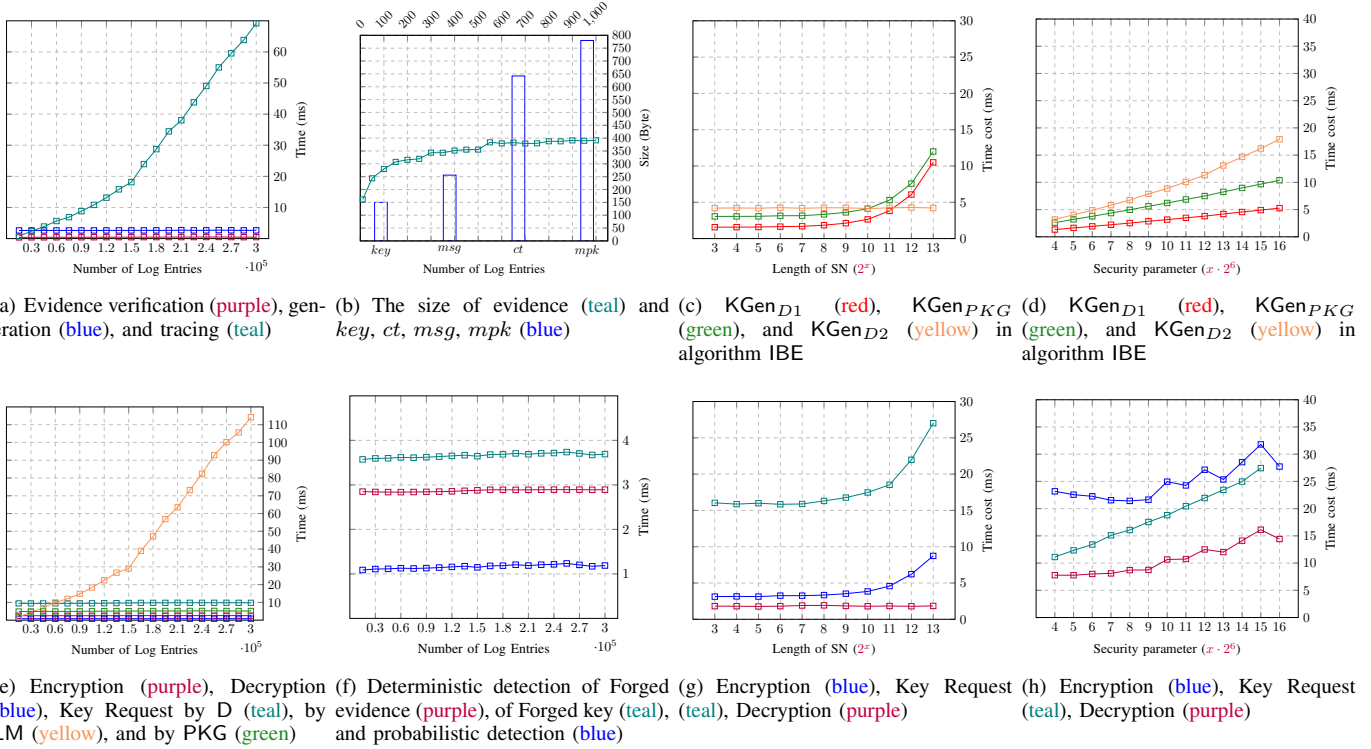


Figure 4: Experimental performance results

integral to the TS, experiences an increment. Consequently, this rise in processing time within the 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 4, when the sequence number length is increased, the running time for the operation key request displays a clear increasing trend. Fortunately, the impact on the running time for encryption and decryption is relatively minimal.

### B. 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 that the evidence size will occupy a constant amount of storage.

## X. DISCUSSION

We first clarify the difference (see Table II) between the concepts of access control, traceability, and accountability and then emphasize the potential applications.

### A. 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.

In the context of “accountable decryption”, the access control defines the decryption time window and specifies the legitimate decryption timeframe, such as permitting the decryptor to decrypt between 3:00 PM and 5:00 PM. Traceability underscores the recording of the specific decryption time to ensure a traceable trail. This means the system accurately records a timestamp for each decryption, facilitating subsequent checks and audits. Accountability focuses on verifying whether decryption occurred within the specified time frame. If decryption takes place outside the permitted timeframe, punitive measures should be applied to ensure accountability for the decryptor’s actions.


**Table II:** 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 like the billing department.
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 or violations.	A nurse has improperly accessed a patient's records in other departments. Accountability requires he/she to provide valid reasons; otherwise, he/she will be sued and punished.

### B. Potential Applications

 *Accountable electronic protected health information.* Electronic protected health information (ePHI) [7] refers to any health information that can identify a patient. Maintaining privacy in the management of ePHI is tricky. Sharing ePHI to all doctors increases the potential risk of information leakage. Among these doctors, there may be potential wrongdoers who exploit ePHI for profit. Conversely, sharing only relevant ePHI with specialized doctors, such as allowing dentists to access only dental medical history without accessing family medical history, contributes to privacy protection. However, in some situations, such as when a patient requires emergency treatment, dentists may miss the optimal rescue window due to their inability to access comprehensive case information. Balancing privacy and data accessibility is challenging and requires careful consideration.

Our system provides a potential solution to address the above challenges. Our solution 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. Encryptors encrypt their data and store it in the hospital's cloud server (trustee TS), making it inaccessible to general practitioners (e.g., dental consultants). However, in certain cases, such as when a patient requires emergency treatment, dentists (i.e., D) can bypass the access control rules and access the ePHI. After that, a designated auditor J (e.g., regulatory authorities or the patient) verifies and accesses the audit trail to monitor whether patient data are being accessed for the right purpose. Thus, we achieve a balance between privacy and data accessibility.

 *Accountable warrant execution.* It is common for government and law enforcement agencies to access citizens' encrypted data through court-issued warrants [40][41]. Unfortunately, these warrants may be abused or misused by malicious officials who exceed their authorized scope, potentially compromising users' sensitive information. Our solution fits in this scenario and prevents data misuse.

In precise terms, when an individual (E) is under accusa-

tion, they encrypt their personal details into a confidential document. Law enforcement (D) may need to access this record without legal consent, posing a dilemma of potential overreach. To oversee the data access behavior, a jury (J) comprising impartial members conducts a post hoc investigation, assessing whether agency officials exceeded authorized boundaries in accessing the individual's information. Thus, the system enables public verification of law agencies' decryption, ensuring the safe and legal use of warrants.

## XI. RELATED WORK

Trusted hardware has been proposed as an effective tool for achieving accountability protocols. Pasture et al. [42] 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 [4] 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 [25] implemented Ryan's protocol using Intel SGX [18], ensuring that the encryptor can detect every decryption. In this scheme, the decryptor can only decrypt the ciphertext after updating the Merkle tree in logs. Liang et al. [43] proposed a decentralized accountable system for healthcare data using Intel SGX and blockchain. Luo et al. [44] 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 [5] employed a TEE-based confidential smart contract to make decryption accountable, where the transaction is used as evidence to trace a user's decryption.

**Table III:** Competitive hardware-enabled solutions

Projects	Policy-making	Action-detecting	Action-feedback	Feedback-reaction	Assumption	TEE Inspect	Accountability role	Accountability operation
Pasture et al. [42]	✗	✓	✗	✗	full	✗	PKG	Keygen
Ryan's [4]	✗	✗	✓	✗	full	✗	User	Enc/Dec
Severinsen's [25]	✗	✓	✗	✗	full	✗	User	Enc/Dec
Liang et al. [43]	✓	✗	✓	✗	full	✗	PKG	Keygen
Luo et al. [44]	✓	✗	✓	✗	full	✗	User	KeyExchange
Fialka [5]	✓	✗	✓	✗	full	✗	User	Dec
<b>PORTEX</b>	✓	✓	✓	✓	semi	✓	User/CR	Keygen/Dec

Different from prior works that merely provide prototypes, our work formally defines the security properties of accountable decryption and proves the correctness of our scheme. 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 com-

promised TEEs. This is achieved by introducing a detection algorithm as part of our solution.

## XII. 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] Joan Feigenbaum, Aaron D Jaggard, Rebecca N Wright, et al. Accountability in computing: concepts and mechanisms. *Foundations and Trends® in Privacy and Security*, 2(4):247–399, 2020.
- [2] Giampaolo Bella and Lawrence C Paulson. Accountability protocols: Formalized and verified. *ACM Transactions on Information and System Security (TISSEC)*, 9(2):138–161, 2006.
- [3] Joan Feigenbaum, James A Hendler, Aaron D Jaggard, Daniel J Weitzner, and Rebecca N Wright. Accountability and deterrence in online life. In *Proceedings of the International Web Science Conference (WEBSCI)*, pages 1–7, 2011.
- [4] Mark D Ryan. Making decryption accountable. In *Cambridge International Workshop on Security Protocols*, pages 93–98. Springer, 2017.
- [5] Rujia Li, Qin Wang, Feng Liu, Qi Wang, and David Galindo. An accountable decryption system based on privacy-preserving smart contracts. In *International Conference on Information Security (ISC)*, pages 372–390. Springer, 2020.
- [6] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel Weitzner. Practical accountability of secret processes. In *USENIX Security Symposium (USENIX Sec)*, pages 657–674, 2018.
- [7] Health insurance portability and accountability act. <https://hipaa.yale.edu/security/break-glass-procedure-granting-emergency-access-critical-epi-systems>, 2004.
- [8] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *ACM SIGOPS Operating Systems Review (OSR)*, 41(6):175–188, 2007.
- [9] Daniel J Weitzner, Harold Abelson, Tim Berners-Lee, Joan Feigenbaum, James Hendler, and Gerald Jay Sussman. Information accountability. *Communications of the ACM (CACM)*, 51(6):82–87, 2008.
- [10] Helen Nissenbaum. Accountability in a computerized society. *Science and Engineering Ethics*, 2:25–42, 1996.
- [11] Rajashekar Kailar. Accountability in electronic commerce protocols. *IEEE Transactions on Software Engineering (TSE)*, 22(5):313–328, 1996.
- [12] 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.
- [13] Ahto Buldas, Helger Lipmaa, and Berry Schoenmakers. Optimally efficient accountable time-stamping. In *International Workshop on Public Key Cryptography (PKCW)*, pages 293–305. Springer, 2000.
- [14] Aydan R Yumerefendi and Jeffrey S Chase. Strong accountability for network storage. *ACM Transactions on Storage (TOS)*, 3(3):11–es, 2007.
- [15] Ruth W Grant and Robert O Keohane. Accountability and abuses of power in world politics. *American Political Science Review*, 99(1):29–43, 2005.
- [16] Joan Feigenbaum, Aaron D Jaggard, and Rebecca N Wright. Towards a formal model of accountability. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 45–56, 2011.
- [17] B Lampson. Notes for a presentation entitled “accountability and freedom,.” <http://research.microsoft.com/en-us/um/people/blampson/slides/AccountabilityAndFreedom.ppt>, 2005.
- [18] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016(86):1–118, 2016.
- [19] Sandro Pinto and Nuno Santos. Demystifying ARM Trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.
- [20] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *European Conference on Computer Systems (EuroSys)*, pages 1–16, 2020.
- [21] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.
- [22] Andreas Kogler, Daniel Gruss, and Michael Schwarz. Minefield: A software-only protection for SGX enclaves against DVFS attacks. In *USENIX Security Symposium (USENIX Sec)*, 2022.
- [23] Fred Cohen. The use of deception techniques: Honeypots and decoys. *Handbook of Information Security*, 3(1):646–655, 2006.
- [24] 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 *Network and Distributed System Security Symposium (NDSS)*, 2020.
- [25] Kristoffer Myrseth Severinsen. Secure programming with intel SGX and novel applications. Master’s thesis, 2017.
- [26] André Martin, Cong Lian, Franz Gregor, Robert Krahn, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. Adam-cs: Advanced asynchronous monotonic counter service. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 426–437. IEEE, 2021.
- [27] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. Iron: Functional encryption using Intel SGX. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 765–782, 2017.
- [28] Andreas Haeberlen. A case for the accountable cloud. *ACM SIGOPS Operating Systems Review (OSR)*, 44(2):52–57, 2010.
- [29] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *Proceedings of the Annual Cryptology Conference (CRYPTO)*, pages 369–386. Springer, 2014.
- [30] 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 Symposium (USENIX Sec)*, pages 699–716, 2021.
- [31] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Annual International Cryptology Conference (CRYPTO)*, pages 213–229. Springer, 2001.
- [32] Benoît Libert and Damien Vergnaud. Towards black-box accountable authority IBE with short ciphertexts and private keys. In *International Workshop on Public Key Cryptography (PKCW)*, pages 235–255. Springer, 2009.
- [33] Ben Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.
- [34] Intel SGX SDK. <https://software.intel.com/en-us/sgx-sdk>.
- [35] The pairing-based cryptography library. <https://crypto.stanford.edu/pbc/>.
- [36] MerkleCpp library. <https://github.com/merklecpp/merklecpp>.
- [37] OpenSSL: Cryptography and ssl/tls toolkit. <https://www.openssl.org/>.
- [38] Intel SGX SSL. <https://github.com/intel/intel-sgx-ssl>.
- [39] Drogon: A C++14/17-based HTTP application framework. <https://github.com/an-taodrogon>.
- [40] 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.
- [41] Matthew Green, Gabriel Kaptchuk, and Gijs Van Laer. Abuse resistant law enforcement access systems. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 553–583. Springer, 2021.
- [42] Ramakrishna Kotla, Thomas L. Rodeheffer, Indrajit Roy, Patrick Stuedi, and Benjamin Wester. Pasture: Secure offline data access using commodity trusted hardware. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- [43] Xueping Liang, Sachin Shetty, Juan Zhao, Daniel Bowden, Danyi Li, and Jihong Liu. Towards decentralized accountability and self-sovereignty in healthcare systems. In *International Conference on Information and Communications Security (ICICS)*, pages 387–398. Springer, 2017.
- [44] Yili Luo, Jia Fan, Chunhua Deng, Yixin Li, Yue Zheng, and Jianwei Ding. Accountable data sharing scheme based on blockchain and SGX. In *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 9–16. IEEE, 2019.

- [45] Michael Sztyldo. Merkle tree traversal in log space and time. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 541–554. Springer, 2004.
- [46] Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium (USENIX Sec)*, pages 317–334, 2009.
- [47] Mark D Ryan. Enhanced certificate transparency and end-to-end encrypted mail. *Cryptology ePrint Archive*, 2013.
- [48] Vipul Goyal. Reducing trust in the PKG in identity based cryptosystems. In *Annual International Cryptology Conference (CRYPTO)*, pages 430–447. Springer, 2007.
- [49] Vipul Goyal, Steve Lu, Amit Sahai, and Brent Waters. Black-box accountable authority identity-based encryption. In *ACM Conference on Computer and Communications Security (CCS)*, pages 427–436, 2008.
- [50] 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.
- [51] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory (TOIT)*, 31(4):469–472, 1985.
- [52] Niels Ferguson and Bruce Schneier. *Practical Cryptography*, volume 141. Wiley New York, 2003.

## APPENDIX A. TRUSTED HARDWARE FORMULATION

**Remote attestation.** Remote attestation is a mechanism that enables remote parties to demonstrate that they are operating a legitimate TEE. In essence, remote attestation is an interactive procedure between the verifier and the attestation service provided by the hardware manufacturer. It first takes a measurement of the data and software running in the TEE and then cryptographically signs it using a hardware-based secret key. Subsequently, the hardware manufacturer, leveraging their secret knowledge of the hardware keys established during manufacturing, verifies the measurement and signature. We formalized remote attestation as follows.

**Definition 9** (Remote attestation unforgeability, RemAttUnf). *Considering that an adversary  $\mathcal{A}$  and a challenger  $\mathcal{C}$  are playing the following game.*

$G_{\text{RemAttUnf}}(\lambda)$
1 : $\mathcal{C}$ runs $pmshw \leftarrow \text{HW.Setup}(\lambda)$
2 : $\mathcal{C}$ initiates a list $\mathcal{L}_q \leftarrow \{\}$
3 : $\mathcal{A}$ runs $hdl \leftarrow \text{HW.Load}(pmshw, Q)$
4 : $\mathcal{A}$ runs $q \leftarrow \text{HW.Run}(hdl, in)$ on any input
5 : $\mathcal{A}$ adds $\mathcal{L}_q \leftarrow \mathcal{L}_q \cup q$
6 : $\mathcal{A}$ outputs $q^* \notin \mathcal{L}_q$

The advantage  $\text{Adv}_{\mathcal{A}, \text{HW}}^{\text{RemAttUnf}}(\lambda)$  of  $\mathcal{A}$  winning the game is

$$\text{Adv}_{\mathcal{A}, \text{HW}}^{\text{RemAttUnf}}(\lambda) = \Pr[\text{HW.VerifyQuote}(pmshw, q^*) = \text{true}].$$

HW is RemAttUnf secure if for all p.p.t. adversaries  $\mathcal{A}$ , there exists a negligible function  $\text{negl}(\lambda)$  such that  $\text{Adv}_{\mathcal{A}, \text{HW}}^{\text{RemAttUnf}}(\lambda) < \text{negl}(\lambda)$ .

**Integrity of TEE code.** TEE creates a secure, isolated area that can run the application and handle sensitive data, which 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 [34]. Thus, the execution of identical loaded code with congruent inputs shall yield consistent outputs.

**Definition 10** (TEE code unchangeability, CodeUncy). *Considering that an adversary  $\mathcal{A}$  and a challenger  $\mathcal{C}$  playing the following game.*

$G_{\text{CodeUncy}}(\lambda)$
1 : $\mathcal{C}$ Run $pmshw \leftarrow \text{HW.Setup}(1^\lambda)$
2 : $\mathcal{C}$ and initialize $\mathcal{O} := \emptyset$ .
3 : $\mathcal{A}$ runs $hdl \leftarrow \text{HW.Load}(pmshw, Q)$
4 : $\mathcal{A}$ runs $q \leftarrow \text{HW.Run}(hdl, in)$ outputting $opt$ .
5 : $\mathcal{A}$ adds $opt$ into $\mathcal{O}$
6 : repeat step 3-5

The advantage  $\text{Adv}_{\mathcal{A}, \text{HW}}^{\text{CodeUncy}}(\lambda)$  of  $\mathcal{A}$  winning the game is

$$\text{Adv}_{\mathcal{A}, \text{HW}}^{\text{CodeUncy}}(\lambda) = \Pr[\text{HW.Run}(hdl, in) \notin \mathcal{O}].$$

HW is CodeUncy secure if for all p.p.t. adversaries  $\mathcal{A}$ , there exists a negligible function  $\text{negl}(\lambda)$  such that  $\text{Adv}_{\mathcal{A}, \text{HW}}^{\text{CodeUncy}}(\lambda) < \text{negl}(\lambda)$ .

**Confidentiality of TEE state.** TEE protects sensitive data and code from unauthorized access and modification by untrusted entities, including the operating system and other applications [18][19][20]. The confidential state can only be accessed by authorized users. We formalized confidentiality through a game. In this game, a probabilistic polynomial time (p.p.t.) adversary  $\mathcal{A}$  tries to distinguish the experiment programs in a real environment and simulated environment. The real-world program works according to the code loaded into TEE. Meanwhile, the *ideal world* is simulated by a p.p.t. simulator  $\mathcal{S}$ .  $\mathcal{S}$  can simulate all entities in the protocol but can only be given  $in$ . The adversary  $\mathcal{A}$  is allowed to access any information about the communication between the entities and any published information (e.g.,  $pmshw$ ). The formal experiment is defined as follows.

$\text{Exp}_{\text{TEE}}^{\text{real}}(\lambda)$	$\text{Exp}_{\text{TEE}}^{\text{ideal}}(\lambda)$
$pmshw \leftarrow \text{Setup}(\lambda)$	$pmshw \leftarrow \mathcal{S}(\lambda)$
$hdl \leftarrow \text{HW.Load}(pmshw, Q)$	$hdl \leftarrow \text{HW.Load}(pmshw, Q)$
$out \leftarrow \text{HW.Run}(hdl, in)$	$out' \leftarrow \mathcal{S}^{\text{HW.Run}(hdl, \{0,1\}^n)}$
<b>return</b> $out$	<b>return</b> $out'$

**Definition 11** (TEE state confidentiality, StaConf). HW is StaConf secure if for all p.p.t. adversaries  $\mathcal{A}$ , there exists a stateful p.p.t. simulator  $\mathcal{S}$ , such that for all  $\mathcal{A}$ ,  $\text{Exp}_{\text{TEE}}^{\text{ideal}}(\lambda)$  and  $\text{Exp}_{\text{TEE}}^{\text{real}}(\lambda)$  is computational indistinguishable, such that  $\text{Adv}_{\mathcal{A}, \text{HW}}^{\text{StaConf}}(\lambda) = \Pr[out = out'] - \frac{1}{2} < \text{negl}(\lambda)$ .

## APPENDIX B. CRYPTOGRAPHIC PRIMITIVES

**Definition 12** (Discrete Logarithm Problem). Let  $N = p \cdot q$  and  $g$  be a primitive root for both  $\mathbb{Z}_p^*$  and  $\mathbb{Z}_q^*$ , where  $p$  and  $q$  are randomly safe primes. Given  $y = g^x \pmod{N}$ , it is computationally intractable to derive  $x$ .

**Merkle tree.** Merkle Tree [45] MT is an optimized form of a hash list, which includes three algorithms.



- **MT.Init( $\lambda$ ):** This algorithm initializes a specific instance of the Merkle tree with a system-level input  $\lambda$ . Assuming the current tree contains a series of leaf nodes denoted as  $(x_1, \dots, x_{n-1})$ .
- **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  $h = \text{MT}(x_1, \dots, x_n)$  and evidence  $\pi$ . The operation represents the process of node merging and updating. The evidence  $\pi$  contains an array of sub-tree hashes that are used for verification.
- **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  $\pi$  and the queried element  $x_n$ , and outputs *true* if the verification succeeds. Otherwise, it outputs *false*.

In an append-only Merkle tree [46][47], 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 [31] consists of the algorithms as follows.

- **IBE.Setup( $\lambda$ ):** The algorithm takes as input the security parameter  $\lambda$ , and outputs a master public key  $mpk$  and a master secret key  $msk$ .
- **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  $key$ .
- **IBE.Enc( $mpk, ID, m$ ):** This algorithm takes as input the master public key  $mpk$ ,  $ID$ , a message  $m$ , and outputs a ciphertext  $ct$ .
- **IBE.Dec( $mpk, sk, ct$ ):** This algorithm takes as input  $mpk$ ,  $sk$ ,  $ct$  and outputs a message  $m$ .

**Definition 13** (Correctness of IBE). *An IBE scheme IBE achieves the correctness if for all  $m \in \mathcal{M}$  and all pairs  $(ID, key) \leftarrow \text{IBE.KGen}(\cdot)$ , it holds that*

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

**Definition 14** (IND-CCA Security of IBE). *Consider an adversary  $\mathcal{A}$  and a challenger  $\mathcal{C}$  playing the following game.*

$G_{\text{IND-CCA}}(\lambda)$

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

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

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

IBE achieves the IND-CCA security, if for all p.p.t. adversaries  $\mathcal{A}$ , there exists a negligible function  $\text{negl}(\lambda)$  such that  $\text{Adv}_{\mathcal{A}, \text{IBE}}^{\text{G}_{\text{IND-CCA}}}(\lambda) < \text{negl}(\lambda)$ . The details refer to [31].

**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 [48] 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 [49], and subsequent research focused on improving its performance. For example, Libert et al. proposed an A-IBE scheme with shorter ciphertext and private key [32], and Zhao et al. reduced the computation cost of A-IBE [50]. 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  $S$  [51] consists of three algorithms, defined as follows. We require the conventional unforgeability property for signatures.

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

**Definition 15** (EUF-CMA Security of S). *Consider an adversary  $\mathcal{A}$  and a challenger  $\mathcal{C}$  playing the following game.*

$G_{\text{EUF-CMA}}(\lambda)$

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

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

$$\text{Adv}_{\mathcal{A}, S}^{\text{G}_{\text{EUF-CMA}}}(\lambda) = \Pr[S.\text{Verify}(vk, \sigma^*, m^*) = 1 | m^* \notin \mathbb{Q}]$$

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

**Public key encryption.** The PKE scheme [52] consists of three algorithms, defined as follows.

- **PKE.KGen( $\lambda$ ):** This algorithm takes as input the security parameter  $\lambda$ , and outputs a pair of keys  $(pk, sk)$ .  $pk$  is the public key and  $sk$  is a secret key.

- $\text{PKE.Enc}(pk, m)$ : This algorithm takes as input a public key  $pk$  and a message  $m$ , and outputs a ciphertext  $ct$ .
- $\text{PKE.Dec}(sk, ct)$ : This algorithm takes as input a secret key  $sk$  and ciphertext  $ct$ , and outputs the message  $m$ .

**Definition 16** (IND-CCA2 security of PKE). *Consider an adversary  $\mathcal{A}$  and a challenger  $\mathcal{C}$  playing the following game.*

$\mathcal{G}_{\text{IND-CCA2}}(\lambda)$

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

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

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

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

**Definition 17** (Collision resistance of hash function). *Consider a hash function  $\Pi = (\text{Gen}, \text{Hash})$  and an adversary  $\mathcal{A}$  playing the following game.*

$\mathcal{G}_{\text{Hash-coll}}(\lambda)$

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

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

## APPENDIX C. 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`.

```

1 element_t msk; // master secret key
2 sgx_ec256_private_t sk_sign; // signing key
3 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  $(X, Y, Z[N + 1], h)$ . We assume that CLIENT and TRACER have already obtained the `mpk` when setup. The `dk_t` contains  $(d1, d2, d3)$ , 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 `Merklepp` 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.

```

1 typedef struct mpk_t {
2     element_t X, Y, h, Z[N + 1];} mpk_t;
3 typedef struct dk_t {
4     element_t d1, d2, d3;} dk_t;
5 typedef struct ct_t {
6     element_t c1, c2, c3, c4;} ct_t;
7 class IbeAlgo {
8 public:
9     pairing_t pairing
10    int size_comp_G1, size_comp_G2, size_Zr,
11        size_GT, ...;
12    // a part of variables
13    element_t Hz, t0, theta, ...;
14    // a part of functions
15    void init();
16    void kgen1(int id);
17    ...
18 }

```

**Listing 2:** Code of `ibe.h` Library.

<p><b>IBE.Setup</b>(<math>\lambda, n</math>)</p> <hr/> <ol style="list-style-type: none"> <li>1: select bilinear groups <math>(\mathbb{G}, \mathbb{G}_T, e, p)</math>, where the <math>p &gt; 2^\lambda</math></li> <li>2: <math>x \xleftarrow{\\$} \mathbb{Z}_p^*</math></li> <li>3: <math>msk \leftarrow x</math></li> <li>4: <math>X \leftarrow g^x</math></li> <li>5: <math>g, h, Y \xleftarrow{\\$} \mathbb{G}</math></li> <li>6: <math>Z \leftarrow (Z_0, Z_1, \dots, Z_n) \xleftarrow{\\$} \mathbb{G}^{n+1}</math></li> <li>7: <math>mpk \leftarrow (X, Y, h, Z)</math></li> <li>8: <b>return</b> <math>mpk, msk</math></li> </ol> <p><b>IBE.KGen</b><math>_{D1}(mpk)</math></p> <hr/> <ol style="list-style-type: none"> <li>1: <math>t_0, \theta \xleftarrow{\\$} \mathbb{Z}_p^*</math></li> <li>2: <b>return</b> <math>h^{t_0} \cdot X^\theta</math></li> </ol> <p><b>IBE.KGen</b><math>_{PKG}(msk, ID, C)</math></p> <hr/> <ol style="list-style-type: none"> <li>1: <math>r', t_1 \xleftarrow{\\$} \mathbb{Z}_p^*</math>;</li> <li>2: <math>d'_1 \leftarrow (Y \cdot C \cdot h^{t_1})^{1/x} \cdot H_Z(ID)^{r'}</math></li> <li>3: <math>d'_2 \leftarrow X^{r'}</math></li> <li>4: <math>d'_3 \leftarrow t_1</math></li> <li>5: <math>pkey \leftarrow (d'_1, d'_2, d'_3)</math></li> <li>6: <b>return</b> <math>pkey</math></li> </ol> <p><b>IBE.KGen</b><math>_{D2}(mpk, ID, pkey)</math></p> <hr/> <ol style="list-style-type: none"> <li>1: <math>r'' \xleftarrow{\\$} \mathbb{Z}_p^*</math></li> <li>2: <math>r \leftarrow r' + r''</math></li> <li>3: <math>d_1 \leftarrow \frac{d'_1}{g^\theta} \cdot H_Z(ID)^{r''}</math></li> <li>4: <math>d_2 \leftarrow d'_2 \cdot X^{r''}</math></li> <li>5: <math>d_3 \leftarrow d'_3 + t_1</math></li> <li>6: <math>key \leftarrow (d_1, d_2, d_3)</math></li> <li>7: checks <math>e(d_1, X) = e(Y, g) \cdot e(h, g)^{d_3} \cdot e(H_Z(ID), d_2)</math></li> <li>8: <b>return</b> <math>key</math></li> </ol> <p><b>H</b>(<math>a, b</math>)</p> <hr/> <ol style="list-style-type: none"> <li>1: <math>c \leftarrow a b</math></li> <li>2: <b>return</b> <math>\Pi.Hash^s(c)</math></li> </ol>	<p><b>IBE.Enc</b>(<math>mpk, ID, m</math>)</p> <hr/> <ol style="list-style-type: none"> <li>1: <math>s \xleftarrow{\\$} \mathbb{Z}_p^*</math></li> <li>2: <math>C_1 \leftarrow X^s</math></li> <li>3: <math>C_2 \leftarrow H_Z(ID)^s</math></li> <li>4: <math>C_3 \leftarrow e(g, h)^s</math></li> <li>5: <math>C_4 \leftarrow m \cdot e(g, Y)^s</math></li> <li>6: <math>ct \leftarrow (C_1, C_2, C_3, C_4)</math></li> <li>7: <b>return</b> <math>ct</math></li> </ol> <p><b>IBE.Dec</b>(<math>mpk, key, ct</math>)</p> <hr/> <ol style="list-style-type: none"> <li>1: <math>(d_1, d_2, d_3) \xleftarrow{parse} key</math></li> <li>2: <math>m \leftarrow C_4 \cdot \left( \frac{e(C_1, d_1)}{e(C_2, d_2) \cdot C_3^{d_3}} \right)^{-1}</math></li> <li>3: <b>return</b> <math>m</math></li> </ol> <p><b>MT.Init</b>(<math>\lambda</math>)</p> <hr/> <ol style="list-style-type: none"> <li>1: <math>\mathcal{LOG} \leftarrow \{\}</math></li> <li>2: <math>vk_{LM}, sk_{LM} \leftarrow S.KGen(\lambda)</math></li> <li>3: <b>return</b> <math>vk_{LM}, sk_{LM}</math></li> </ol> <p><b>MT.Insert</b>(<math>T</math>)</p> <hr/> <ol style="list-style-type: none"> <li>1: <math>n \leftarrow \text{number of leaf nodes}</math></li> <li>2: <i>insert</i> <math>T</math> to <math>\mathcal{LOG}</math></li> <li>3: <math>H_{old} \leftarrow H(0, n-1), H_{new} \leftarrow H(0, n)</math></li> <li>4: <math>\rho \leftarrow \{\text{hash nodes covering } [0, n]\}</math></li> <li>5: <math>\varepsilon \leftarrow \rho \cup H(n, n)</math></li> <li>6: <math>\pi \leftarrow (N, H_{new}, H_{old}, \rho, \varepsilon)</math></li> <li>7: <b>return</b> <math>\pi</math></li> </ol> <p><b>MT.Verify</b>(<math>\pi</math>)</p> <hr/> <ol style="list-style-type: none"> <li>1: <math>(N, H_{new}, H_{old}, \rho, \varepsilon) \xleftarrow{parse} \pi</math></li> <li>2: <math>H_{poe}, H_{pop} \leftarrow \phi</math></li> <li>3: <b>foreach</b> <math>H_i \in \varepsilon</math></li> <li>4:     <math>H_{poe} \leftarrow \Pi.Hash(H_i, H_{poe})</math></li> <li>5: <b>endforeach</b></li> <li>6: <b>foreach</b> <math>H_j \in \rho</math></li> <li>7:     <math>H_{pop} \leftarrow \Pi.Hash(H_j, H_{pop})</math></li> <li>8: <b>endforeach</b></li> <li>9: <b>return</b> <math>(\Pi.Hash(N) \in \varepsilon \text{ and } H_{poe} = H_{old} \text{ and } H_{pop} = H_{new})</math></li> </ol>
--	--

**Figure 5:** IBE algorithm and evidence management algorithm