CrudiTEE: A Stick-and-Carrot Approach to Building Trustworthy Cryptocurrency Wallets with TEEs

Lulu Zhou
Yale University, USA

Zeyu Liu
Yale University, USA

Fan Zhang
Yale University, USA

Michael K. Reiter
Duke University, USA

Abstract

Cryptocurrency introduces usability challenges by requiring users to manage signing keys. Popular signing key management services (e.g., custodial wallets), however, either introduce a trusted party or burden users with managing signing key shares, posing the same usability challenges. TEE (Trusted Execution Environment) is a promising technology to avoid both, but practical implementations of TEEs suffer from various side-channel attacks that have proven hard to eliminate.

This paper explores a new approach to side-channel mitigation through *economic incentives* for TEE-based cryptocurrency wallet solutions. By taking the cost and profit of side-channel attacks into consideration, we designed a Stick-and-Carrot-based cryptocurrency wallet, CrudiTEE[1], that leverages penalties (the stick) and rewards (the carrot) to disincentivize attackers from exfiltrating signing keys in the first place. We model the attacker's behavior using a Markov Decision Process (MDP) to evaluate the effectiveness of the bounty and enable the service provider to adjust the parameters of the bounty's reward function accordingly.

2012 ACM Subject Classification Security and privacy → Authorization; Security and privacy → Side-channel analysis and countermeasures

Keywords and phrases Cryptocurrency wallet, blockchain

Digital Object Identifier 10.4230/LIPIcs.AFT.2024.5

Related Version *Full Version*: `https://arxiv.org/abs/2407.16473`

Supplementary Material *Software*: `https://github.com/luluzhou1/MDP_for_Bounty_Evaluation`

Funding This work was funded in part by NSF grant 2207214 and an Ethereum Academic Grant.

1 Introduction

As cryptocurrencies [8, 59] gain popularity, more and more people use cryptographic signatures as a way to authorize transactions. Unfortunately, signing key management has long been a notoriously hard problem. With inexperienced users often struggling with lost or leaked keys, a natural tendency is to outsource the task to specialized service providers. For example, 11% of the entire cryptocurrency marketization is stored in custody by a single service provider (Coinbase [61]). This is undesirable security-wise, as the secrecy of keys (thus the safety of the funds) relies on the trustworthiness of a centralized party.

[1] Crudite is a salad with carrots and (other) vegetable sticks.

To provide stronger security guarantees (and to reduce liability), a cryptocurrency wallet service provider can generate users' signing keys in Trusted Execution Environments (TEEs, such as Intel SGX [3, 37], AMD SEV [2], Nvidia H100 [28]) and serve signing requests in TEE without ever seeing the signing keys in plaintext. However, the naive adoption of TEEs does not provide a meaningful secrecy guarantee *to users*, because the service provider may be able to exfiltrate signing keys through side-channel attacks [43]. While side-channel mitigation has been extensively studied in the literature (e.g., see [52] for a survey), side channels are notoriously hard to eliminate, due to the complexity of modern processor design (e.g., TEEs often share physical resources with untrusted processes, such as caches).

Our work is motivated by the observation that the operator of TEEs is the primary actor capable of mounting side-channel attacks, since most attacks [57, 43, 34, 49, 35, 42] require root access to the host. For wallet key management services, the TEE operator is the service provider. This observation gives us additional leverage to prevent side-channel attacks, as the service provider can be held responsible (using techniques to be presented later) if a wallet key is leaked or accessed without user authorization. Using a proper penalty mechanism, we can eliminate the service provider's gains from a successful side-channel attack, thus removing the incentive to attack in the first place.

With the TEE operator striving to avoid key leakage, the possibility of side-channel attacks by non-local, unprivileged attackers is significantly reduced (e.g., the service provider is motivated to employ heightened security measures). To further discourage such attacks, our idea is to reward the attackers for partial success. For example, if a signing key is distributed cross N TEEs using secret sharing, we give the attacker a substantial reward if he successfully exfiltrated *any* share. With a proper reward function, this early reward can serve as a strong incentive for the attacker to *stop early*, giving the system administrator time to react to partial compromise before a full key is exfiltrated.

1.1 CrudiTEE: The Cryptocurrency Wallet with Stick and Carrot

Based on the above two principles, we propose CrudiTEE, a TEE-based cryptocurrency wallet that can defend against TEE side channels by privileged and unprivileged attackers, using penalties (stick) and rewards (carrot), respectively. Furthermore, CrudiTEE strives to achieve user-friendliness (i.e., users do not need to store keys locally). CrudiTEE first requires that the signing keys be generated inside TEE and never exported in plaintext. Assuming correct implementation, this implies that signing key leakage is impossible except for through side-channel attacks.

We classify potential actors capable of mounting side channel attacks into *insider attackers* and *outsider attackers*. The insiders are privileged attackers, such as service providers, who have full control over the TEE including physical access. Insiders have powerful attacking capabilities required by most side-channel attacks (such as root privilege) like the ones needed in [31, 20, 47]. In contrast, the outsiders are all the attackers who can exfiltrate the secrets in the TEEs only through less-privileged means like remote time-based attacks [32, 12, 1]. We refer readers to Section 2.2 for more examples. As introduced above, CrudiTEE consists of the stick (penalties), to discourage insider attackers, and the carrot (rewards), to encourage outsider attackers to stop early.

Note that to perform such punishment or distribute the bounty, we need an automated but also trustworthy and publicly accessible mechanism. Smart contracts [59](autonomous programs executed on blockchains) are the perfect tool for this purpose. Thus, below, when discussing the stick and the carrot, we use the smart contract as an important building block.

1.1.1 The stick

Due to the power of the service provider, preventing it from mounting side channels via a technical way seems infeasible. Instead, CrudiTEE requires the service provider to put down *collateral*, which will be confiscated if signing keys safeguarded by the TEEs are used for unauthorized signatures or if legit service requests from users are denied.

To realize the stick of CrudiTEE, the key is to enable a user to generate publicly verifiable proof if her TEE-generated keys are illegally accessed. First, as mentioned, raw keys stay in the TEE and are never exported outside. Second, each key corresponds to a wallet owner and can only be used by the owner through well-defined APIs (e.g., an API could allow the owner to sign messages with the key using a carefully implemented signature algorithm). Third, to access a key, a signed authorization from its owner must be present and checked by TEEs, thus making the authorization process *accountable* (i.e., if the user disputes a signature, the service provider can present proof that the signature was authorized by the user). Users can verify TEE attestations to ensure the prerequisites are met before signing up for the service.

In order not to burden the user with signing key management while making the authorization process accountable, we use the OAuth protocol (Section 3.3). The token signed by the OAuth provider is used as proof of authorization.

The service provider sets up a smart contract to implement the insurance (denoted $\mathsf{SC}_{\mathrm{ins}}$) with the following logic and makes an initial deposit. If a user discovers any unauthorized signature, she can submit a request to $\mathsf{SC}_{\mathrm{ins}}$. The service provider must prove that the user had authorized such key use within a specific period. Failing to provide such proof results in the insurance smart contract automatically compensating the user.

1.1.2 The carrot

Without the help of any insiders, outside attacks become unlikely, but still not impossible. To limit potential exposure to external attacks, we employ the threshold signing protocol such as [29], where the signing key is stored as key shares across multiple independent TEEs (e.g., hosted in different clouds) and refresh secret shares periodically. This way, even if an outside attacker can exfiltrate a few shares, he needs all shares to exfiltrate the entire key. However, the security of such proactive secret sharing method as a defense is "black or white" — unless the attacker can break a sufficient number of TEEs and cause a catastrophic breach, partial breaches cannot be detected and therefore cannot inform the service provider to take proper action to prevent those catastrophic breaches. By exploiting economic incentives, we can elicit such information from the attacker. Specifically, CrudiTEE enhances a proactive secret-sharing scheme with an alerting mechanism so that when partial breaches happen (e.g., TEEs deployed in one cloud are vulnerable, but not others), the attacker is encouraged to alert the service provider in exchange for a *bounty*. This allows the service provider to take proper action before full breaches happen.

Designing a bounty reward function that induces the desired behavior of the attackers is the main technical challenge. Specifically, we aim to formulate a reward function that motivates attackers to promptly alert the service provider without generating any illegal signature or selling the acquired signing key shares, while minimizing the defender's cost (i.e., the service provider's cost). We employ a 2-step methodology in the reward function design: we start with the attacker with a fixed known cost first and then deal with the one whose attacking process is non-deterministic and whose cost cannot be accurately estimated.

Step 1: We provide the following toy example to illustrate the challenge in reward function design under a deterministic setting. We start with a key (worth \$3 in total) stored as three secret shares, each of which is worth \$1 (assuming a share can be sold on the market for \$1). To steal one share, the attacker's cost is \$0.8. Furthermore, assume that \$0.01 is the smallest unit of money for simplicity. Without a bounty, the attacker will keep attacking until he gets 3 shares and sells them on the market for \$3, making a profit of \$0.6.

To protect against such an attacker, there are two naive but natural solutions. The first solution is to simply have the reward function be a constant function of \$3.01 (i.e., the attacker obtains \$3.01 for any amount of shares he steals). In this case, an attacker always submits the share as soon as he obtains the first share, but then the defender costs more than the key value itself. The second solution is setting the function to be \$1.01 per share (i.e., a function linear in the number of shares). However, in this case, an attacker would instead try to obtain all three shares and claim a total reward of \$3.03, which costs even more.

The optimal solution is to set the reward function to be a constant function of \$1.41 (i.e., the attacker is awarded with \$1.41 for submitting any amount of shares): the attacker will stop attacking and turn in the key shares whenever he obtains 1 key share, making a profit of \$0.61. This reward function not only encourages the attacker to submit as soon as getting one share but also minimizes the defender's cost. Note that it is indeed the least the service provider can pay, as if the reward is less than \$1.41, the attacker will sell the key for a higher profit instead (assuming w.l.o.g. that the attacker sells the secret when the profit from the bounty is tied with selling the key).

Step 2: The reward function in the toy example, however, is based on a simplified assumption of deterministic attack costs and requires the defender to accurately know the attacker's cost. Our design instead aims to address real-world situations where the attacker's attack process is non-deterministic, and the cost of attacking cannot be accurately known in advance.

To design the reward function in this setting, we first turn the desired properties of the reward function into numerical metrics. Then we capture the non-deterministic attacking process as an "optimal stopping" game and use Markov Decision Process (MDP) to analyze the attacker's optimal strategy. We propose a reward function for non-deterministic attackers and optimize it using the metrics as an objective function, based on the defender's budget and estimation of the attacker's cost and success rate. We further show that the reward function not only has good performance for the attacker with an accurately estimated cost but also for attackers with different costs. We provide the defender with the performance of the optimized reward function for attackers with a wide range of costs and success rates. The defender can use such a strategy to assess how the reward function she obtains performs for a range of attackers. If she is not satisfied with the result, she could raise their budget and generate another function.

To realize the bounty, the service provider creates a smart contract $\mathsf{SC}_{\text{bounty}}$ that accepts proofs of knowledge (PoK) of TEE-managed key shares and remits rewards accordingly. Valid PoK submissions to $\mathsf{SC}_{\text{bounty}}$ raise a flag, pausing operations until the keys are rotated and the flag is reset. To ensure that the attacker did not use the breached key for unauthorized signings, users are requested to check for unauthorized signatures during the shutdown period. If any are found, the attacker's reward is forfeited.

Contribution

We summarize our contributions as follows:

1. We introduce a new approach to building a cryptocurrency wallet: CrudiTEE that leverages *economic incentives* to defend against side-channel attacks from insiders and outsiders.

2. CrudiTEE involves a novel automatic insurance system (Section 5), allowing users to receive compensation if their wallet signing key is used for signing transactions without their authorization.
3. We develop a reward function for the bounty in CrudiTEE (Section 6) that encourages attackers to submit key shares to the bounty immediately while minimizing the defender's cost. We use the Markov Decision Process (MDP) to model the non-deterministic nature of side-channel attacks and optimize the reward function against numerical metrics. We evaluate and show the optimized reward function is effective not only for attackers with precisely estimated costs but also for attackers with variable costs. The service provider may adjust her budget to cover a wider range of attackers the reward function can effectively defend against based on the evaluation.

2 Related Work

2.1 Cyber Bounty

Setting up bug bounties is a popular way to defend against hackers [36]. However, a fair exchange of bugs and money is difficult without trust. Breidenbach et al. [10] proposed that smart contracts be deployed to guarantee that the attacker gets paid once a valid bug is submitted. Their game-theoretic analysis showed that the attacker is incentivized to submit the bug as soon as possible because of competition from other honest hackers. However, this is not always the case for side-channel attacks: a malicious attacker may be the only one to discover a zero-day[2] side channel. That is why we take the submission time into consideration in our reward function, i.e., to incentivize attackers to submit the leaked signing key (share) immediately upon acquiring it.

2.2 Side Channels

Side-channel attacks against cryptographic systems usually take one of three forms. *Time-driven* side-channel attacks expose key information by monitoring *total* execution times of cryptographic operations with a fixed key, which can reflect interactions among the value of the key, the structure of the cryptographic implementation, and system-level effects such as cache evictions (e.g., [32, 12, 1, 58]). *Trace-driven* side-channel attacks observe a time-series signal reflecting a device's cryptographic operation *throughout its execution*, e.g., by monitoring the device's power draw during the operation (e.g., [31]) or its electromagnetic emanations (e.g., [20, 47]). Finally, in an *access-driven* side-channel attack, the attacker executes a program on the same computer where the cryptographic operation is taking place, using this vantage point to monitor the operation's use of microarchitectural components on the platform (e.g., [45, 27, 26]). Time-driven and trace-driven attacks are largely agnostic to the encapsulation of the cryptographic operation within a TEE. In contrast, much effort has been expended to adapt access-driven attacks to attack a cryptographic operation executed within TEE from outside, with considerable success (e.g., [57, 43, 34]).

Using the terminology of Section 1, we consider *outsiders* to be less privileged and thus limited to time-driven and some access-driven attacks, that can be performed remotely (i.e., without any physical access to the TEE). Any attacks available to an outsider, however, must incur costs to conduct over time, e.g., to achieve and maintain co-residency on the same

[2] A zero-day is a vulnerability in software or hardware that is unknown to its vendor.

physical computer as the victim computation [56] (possibly despite defenses to make this difficult, e.g., [41]) and to perform attack computations. In contrast, *insiders* are permitted to conduct *any* time-driven, trace-driven, or access-driven attacks, and so are considerably more powerful. In particular, we design CrudiTEE in anticipation of insiders capable of extracting keys from TEEs easily. Outsiders, on the other hand, are assumed to require more time and costs to mount their attacks.

2.3 TEE Side-channel Defense

A recent concurrent and independent work, Sting [7], proposes to use SC as a bug bounty, which is set up to encourage anyone who has access to a leaked secret to submit proof. The proof of leakage is acquired in this way: first, a prover-owned TEE generates a secret, without disclosing it to the prover. Second, the secret is directly sent to the secret management service provider (without exposing the secret to the prover). Finally, the prover acquires the secret using a side-channel attack, sends it back to the prover-owned TEE, and gets proof of leakage from the TEE. Sting focuses more on the proof generation rather than the bounty design, however. This is different from our bounty as we encourage attackers (*without* physical access to the machine) to stop recovering the secret and submit a bounty claim without recovering the whole secret via economic incentives.

Numerous techniques other than bug bounty could be applied to side-channel defense, including ORAM [16], code hardening [11], data location randomization [9]. However, defenses introduce performance overheads and usually defend against only specific types of attacks. Another problem is that a service provider might not have enough incentive to apply these defensive technologies expeditiously. Therefore, motivating the service providers to keep their TEEs safe from attack is crucial to the real-world use of TEEs.

2.4 Existing Wallet Solutions

Some companies provide the service like a centralized bank for cryptocurrency [15], holding users' funds in company-owned accounts. Such centralized service deviates from the decentralized nature of cryptocurrency and increases risk to user funds. On the other hand, there are products to enable users to store their signing keys in a protected area of an offline device, named hardware wallet [51]. This approach raises costs and complicates transactions, and users usually have to trust the software provided by the hardware manufacturer for signing transactions. A keyless wallet was constructed using witness encryption [63]. To access the money, the user only needs to provide a short one-time password of 6 alphanumeric characters generated from an offline device. Since Witness Encryption is currently impractical, however, the scheme is largely theoretical.

3 Background and Preliminaries

3.1 Trusted Execution Environments

TEEs (Trusted Execution Environments) are secure and isolated execution environments that provide confidentiality and integrity guarantees and the ability for a party to remotely verify the status of a TEE through remote attestation. Prominent examples of TEEs include Intel SGX [3, 37], AMD SEV [2], and Nvidia H100 [28]. A major practical limitation of TEEs is side channel attacks (Section 2.2) that could break the confidentiality guarantee.

3.2 Smart Contracts

To create elaborate economic incentive structures, CrudiTEE uses smart contracts, autonomous programs running on top of blockchains, to remit payments under specific events. We follow the standard assumption that smart contracts are correct (i.e., the security assumptions required by the blockchain protocol are met) and available (i.e., all parties in our protocols can access the smart contract and request submitted to the smart contract is executed within a time limit).

3.3 OAuth

CrudiTEE uses the OpenID Connect feature in OAuth (Open Authorization) 2.0 [25, 46] to enable users to make signing requests without possessing a signing key. OpenID is an authentication protocol that allows users to use an existing account from an OpenID provider (denoted as "OAuth provider"), such as Google, to authenticate themselves on other applications. Furthermore, during authentication, a user can embed a customized message in the 'nonce' field of the signed ID token [25] (looking ahead, this allows the user to put a description of her request in this field).

3.4 Cryptographic Primitives

We provide a brief description of the threshold signing scheme.

Threshold signature allows $N > 1$ parties to share a secret signing key, such that each party obtains a share of the signing key. Only when m parties owning a sharing, $1 \leq m < N$, together can sign a message. Knowledge of $< m$ shares leaks no information about the secret signing key. Furthermore, when the secret shares are updated to N new shares, even $m_1 < m$ old shares and $m_2 < m$ new shares where $m_1 + m_2 \geq m$ together leak no information about the secret. We use it to allow multiple TEEs to share the signing key, such that only if $\geq m$ shares are leaked, the secret is leaked.

3.5 Markov Decision Process

A Markov decision process (MDP) is a mathematical model that captures decision-making under uncertain situations. A Markov state is a state S_t at time $t > 0$ satisfying $\Pr[S_t|S_{t-1}] = \Pr[S_t|S_{t-1}, \ldots, S_1]$ (i.e., the previous state captures the entire history states). The MDP consists of a sequence of Markov states and an associated state transition matrix. This matrix represents the probabilities of transitioning from one state to another based on the player's actions. The player's optimal strategy in MDP can be computed using tools like [13].

4 Threat Model and Roadmap

4.1 Threat Model

The purpose of the techniques in CrudiTEE is to mitigate the side-channel attacks that break the privacy of the TEEs but not the integrity. We assume TEE integrity (i.e. the data and code in the TEE cannot be modified by any attacker) to hold and remote attestation to be secure, following a common assumption (c.f., [53, 14]), as the attestation key is only used through a limited interface, unlike application-generated secrets. The side-channel attacks that are strong enough to compromise the attestation key [55] are out of scope for this work, as such incidents have historically been rare.

We assume that the integrity and liveness of smart contracts are enforced by the blockchain. Furthermore, we assume the OAuth providers are trusted, but note that any user can choose her own set of OAuth providers to trust (i.e., the user can choose a subset of a predefined set of OAuth providers). Finally, we assume that both the service provider and the outsider attacker are rational entities aiming to maximize their profits. We do not consider non-financial incentives, and the agent who attacks the system as a mere malicious intruder is out of our scope.

4.2 Wallet Design Overview

In our wallet service, each client registered with the wallet service provider has a wallet whose signing key is stored in the service provider's TEE. Our goal is to defend side-channels against such signing keys.

We categorize side-channel attacks into two types: insider attacks, which require physical access and/or root privileges, and outsider attacks which can be executed remotely without such privileges (Section 2.2). In our wallet design, the service provider, who controls the TEEs, is classified as an insider, whereas all other attackers, including users, are categorized as outsiders. We defend the insiders using the insurance (the stick) and the outsiders using the bounty (the carrot).

The side-channel mitigation in CrudiTEE thus consists of three main components:

1. The accountable signing key management service (Section 5.1) enables the users to register for the service and authorize the service provider to sign a transaction when needed.
2. The insurance (Section 5.2) ensures the service provider provides the desired service, and otherwise is punished.
3. The bounty (Section 6) aims to incentivize the outsider attacker to submit the key shares acquired through the remote side channel to the bounty (smart contract) rather than using them to make unauthorized signatures or selling them.

Both the insurance and the bounty are initiated using smart contracts (SC_{ins} and $\mathsf{SC}_{\text{bounty}}$). In addition, to make sure that the service provider answers all the service requests (instead of ignoring those requests), the smart contract $\mathsf{SC}_{\text{avail}}$ is also deployed. During setup, the service provider needs to build the TEE program and publish the attestation. Then, the service provider deploys the aforementioned smart contracts on the blockchain.

To use the service, the user first chooses the OAuth provider(s) she trusts and creates a new account with her OAuth token (signed by that OAuth provider(s)). The service provider will execute the threshold key-generation protocol among the TEEs, register the OAuth account and key mapping, and then provide the public key to the user. It is essential that the signing key is generated within the TEEs and remains within the TEEs (i.e. cannot be exported in plaintext format). This is because if the users learn the key, it becomes ambiguous whether the responsibility for any unauthorized signature lies with the users or the service provider. After the generation of the signing key, a smart contract wallet $\mathsf{SC}_{\text{wallet}}$ will be deployed for the user. $\mathsf{SC}_{\text{bounty}}$ will also be updated so that the new key is also protected by the bounty. The proof-of-publication[3] scheme is employed to ensure that the smart contract update is done properly.

The service provider replies to the user's transaction signing requests with authentication via OAuth providers (Figure 1). The signing is conducted using the threshold signature

[3] Proof of publication is a way for the TEE to verify that a state change is updated on the blockchain.

scheme, with the signing key secret-shared among several TEEs. When the service provider is not responding to a signing request, the user can send the request through $\mathsf{SC}_{\text{avail}}$ and force the service provider to respond. If the user realizes that an unauthorized signature exists, she can submit a claim to SC_{ins} and get compensated (Figure 2).

Finally, if an outsider attacker steals the signing key (shares) from a remote side channel, he can submit it to $\mathsf{SC}_{\text{bounty}}$ and get rewarded based on the submission time and number of shares he submits (Figure 2). Any valid $\mathsf{SC}_{\text{bounty}}$ or SC_{ins} submission will trigger a flag to signify that some of the TEEs have been breached. CrudiTEE requires that all wallet transactions cease until the service provider rotates all the signing keys and clears the flag. If the full key is leaked, the TEE will generate a fresh key pair, update the OAuth account and wallet key mapping, and transfer the money in the smart contract wallet to the new wallet while the red flag is on. Transactions during the red flag period can only be triggered by a message signed by the TEE attestation key. The reward for the attacker will be held for a specified period, during which the user of the affected keys will be asked to check whether there exists any unauthorized transactions and the reward will not be given to the attacker if such transactions are found.

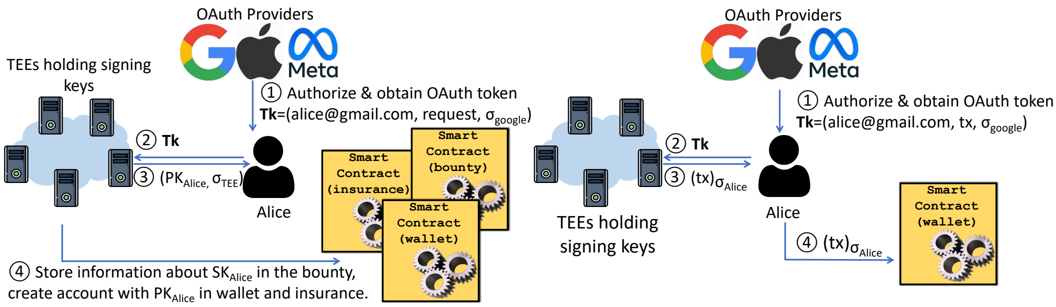
4.3 Reward Function Design Roadmap

The attacker's reward is determined by a reward function designed to incentivize them to claim the bounty immediately upon obtaining a single key share from the TEEs, while minimizing the defender's cost (Section 6.3). Since the reward function design is particularly challenging among other components of the wallet, we discuss our roadmap here. We employ a 2-step methodology here: First, we deal with attackers with known deterministic costs (a simplified case). Then, we employ the ideas from this simplified case together with other more advanced mechanisms to develop the reward function for the attacker with non-deterministic and unknown costs.

In more detail, we begin with a case study assuming the attacker operates under a deterministic cost function known by the defender. However, in the real world, the side-channel attacking process is non-deterministic, and the cost of the attack is hard to estimate accurately. Building on insights gained from the case study, we propose a reward function for attackers with non-deterministic behavior. We model the non-deterministic attacking process as the "optimal stopping" game [54, 50, 24] and employ Markov Decision Processes to calculate the best strategies for the attackers. By translating the desired properties of this reward function into quantitative metrics used as the objective function, we optimize the parameters in the reward function (based on the defender's budget and her estimation of the capability of the attacker). Finally, we evaluate the effectiveness of our proposed reward function when the attacker's ability (parameterized by his cost and success rate) is different from the estimations. Based on the evaluation of the attacker, the defender can further raise her budget and recompute the function to get a more satisfying range of attackers the function can defend against.

5 The Stick

In this section, we first provide more details about the wallet workflow (Section 5.1), which outlines the responsibilities of the service provider. Then, we specify the "stick" part which holds the service provider responsible (Section 5.2).



(a) Registration process. **(b)** Transaction signing process.

Figure 1 Registration and Transaction Signing Workflow

5.1 Authorization and Signing Transactions

We start by elaborating on how we make the authorization of the transactions accountable and describe how a user registers for an account and requests signed transactions.

Accountable authorization: As mentioned in the Section 1.1.1, an authorization process is *accountable* if it leaves a signed evidence that can be used to prove the validity of the signing key usage later. Meanwhile, it should not burden the user with additional key management.

Our solution leverages a feature in OAuth 2.0 called OpenID Connect (OIDC) [46, 25]. Specifically, OIDC-enabled OAuth providers issue signed identity tokens (called `ID_token` [25]) that include a user identifier (such as email addresses) and a nonce set by users. Many mainstream OAuth providers enable the user application to specify the nonce in the ID token (e.g., Google [25], Microsoft[39], etc.).

Every time the signing key is used, we require the user to provide an ID token signed by the OAuth provider(s), which is uniquely linked to that specific signing request by including the request hash in the nonce field. TEE verifies the token of the corresponding OAuth provider(s)' keys accordingly. The public key of the OAuth providers is hardcoded in TEE and verified by the user through attestation. This method not only provides a log-in process that most users are familiar with, but also delegates authorization to a third party (or a set of third parties) that they trust, providing signed OAuth token(s) as proof of authorization.

Registration: As shown in Figure 1 (a), when registering for a new account, the user runs a protocol to determine the future authentication process with the service provider. Specifically, the user first chooses a set of OAuth provider(s) she trusts. Next, she puts the hash of the account registration request (e.g. the hash of "CrudiTEE account registration") in the 'nonce' field of the ID token, authenticates it with the OAuth provider, and asks the OAuth provider to sign it. Then, the user sends the account registration request to the service provider along with the token(s). TEE verifies the token(s) and generates a fresh key pair for signing. The TEE creates a TEE-signed receipt with the newly generated verification key (to verify the signed transactions for this user's wallet) and the OAuth ID(s) associated with it. Lastly, a smart contract wallet is created for the user.

Transaction signing request: As shown in Figure 1 (b), when the user wants to sign a transaction, she generates a signing request. Then, she acquires a signed token from the OAuth provider(s) with the hash of the transaction included in the token(s). Once receiving the signing request and token(s), the service provider should input it into the TEEs. The TEE will check the validity of the request by verifying the token(s) and respond accordingly (we discuss how to enforce the TEEs to respond in Section 5.2.1). If the request is valid, the TEE will reply with the signature of the transaction, generated with the signing key associated with the user's OAuth ID(s). If not, the TEE will reply with a message saying that the request is invalid, signed with its attestation key. We require TEEs to store the (valid) tokens and requests in case of any future insurance claim (Section 5.2.2). The signed transaction will be submitted by the user to the wallet smart contract $\mathsf{SC}_{\text{wallet}}$. The wallet smart contract will check the signature and execute the transaction.

Threshold signing: CrudiTEE use a threshold signature scheme (e.g., [22]) for singing. Specifically, the key-management service provider secret-shares each key into N secret shares using a (m, N)-threshold-signature scheme (where $m \leq N$), stores them in independent TEEs, and rotates them every T units of time. This approach not only serves to complicate the execution of side-channel attacks but also establishes the foundation for the bounty scheme described in Section 6.

5.2 The stick: hold service provider responsible

Based on the accountable signing process described in the previous subsection, the "stick" aims to establish mechanisms to punish the service provider when it misbehaves. The goal is that *any* rational service provider would not choose to misbehave (e.g., steal the secret and produce an unauthorized signature).

5.2.1 Ensure Availability of TEE

We start by discussing how to ensure that service providers process requests using TEE (with the expected inputs), guaranteeing TEE's availability [4]. The service provider sets up $\mathsf{SC}_{\text{avail}}$ and makes the initial deposit. If the service provider refuses to process a signing request directly submitted to the service provider, the user submits the request to $\mathsf{SC}_{\text{avail}}$. The service provider monitors the SC, processes any request from the SC, and forwards the request to the TEE. The TEE then generates a reply, which is either the requested signature or indicates that the request is invalid. The reply, along with the user's request, must be signed by the TEE's attestation key. After receiving the reply, $\mathsf{SC}_{\text{avail}}$ checks whether the reply is signed by the TEE's attestation key and the request is included in the signed message.[5] If it is, $\mathsf{SC}_{\text{avail}}$ records the reply. If the service provider does not submit a valid reply within a time limit, its deposit gets burnt (destroyed). [6]

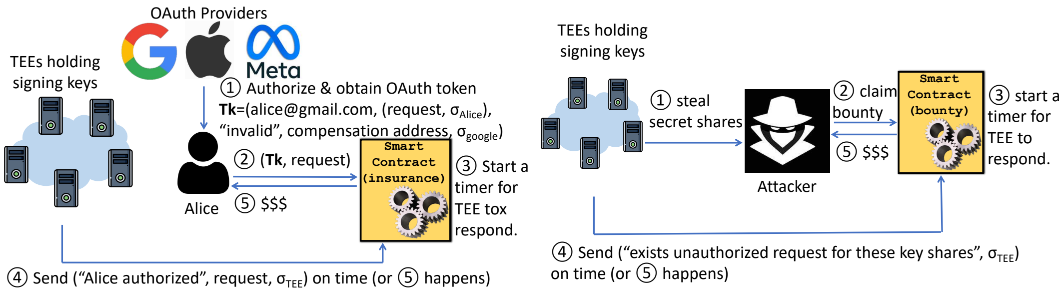
5.2.2 Insurance for unauthorized transactions

In this part, we develop a mechanism that enables users to report unauthorized transactions. As shown in Figure 2 (a), the user submits the signature to request a message, signed by the TEE's attestation key, stating that the signature is authorized by the user. When the service provider is unable to provide such a message, the user is automatically compensated. Since the user initiates the insurance claim, they are responsible for monitoring transactions

[4] The idea of using incentives to make a service available is not new, though. A similar method is used in blockchain Layer2 to prevent transaction censorship [5].

[5] Attestation key is hardcoded to the smart contract.

[6] Note that one may consider a DoS-attack: initiating many small transactions using $\mathsf{SC}_{\text{avail}}$. To avoid this, the service provider can set a corresponding transaction fee to use $\mathsf{SC}_{\text{avail}}$ paid by the user. If the user, however, needs to use such a service, the user may consider the service provider as malicious, thus withdrawing all the money and stop using the service. Thus, a rational service provider would avoid letting the user make transactions via $\mathsf{SC}_{\text{avail}}$.



(a) Insurance Claim through SC_{ins}. **(b)** Bounty claim through the $\mathsf{SC}_{\text{bounty}}$.

Figure 2 Insurance and bounty workflow

and submitting complaints for unauthorized transactions, similar to most systems based on staking and slashing [33].

We instantiate the insurance using a smart contract (SC_{ins}). This smart contract specifies the necessary ground truth requirements, such as the attestation key of the TEEs, and the conditions under which users are eligible for compensation. A predefined quantity of deposits is deposited in it, serving as potential compensation for the user.

An insurance claim is initiated by the submission of an unauthorized transaction to SC_{ins} together with the proof of ownership of the key. The proof of ownership is a message stating the ownership of the key signed by the TEE, which could be requested using the user's OAuth token. SC_{ins} checks whether the claim for the transaction has not yet been made before. If yes, the claim will be rejected. The service provider monitors SC_{ins} and sends the request to the TEE once it is published on the blockchain. The TEE looks for the authentication token(s) associated with this request (recall that the valid requests are stored). If no valid token(s) in question are found, the TEE will sign a message stating that the signature was unauthorized with its attestation key. Otherwise, a message stating that the signature was authorized will be signed. The service provider submits the reply to SC_{ins}. SC_{ins} checks whether the message signed by the TEE attestation key states that the signature was authorized. If not, SC_{ins} compensates the user (for some predetermined value that depends on the application) and records this claim (e.g., on the chain) for future reference. If the service provider fails to submit the requisite proof within the specified timeframe, the user automatically gets compensated from the smart contract.

Security analysis: We briefly analyze how the initial goal was achieved with the design of the "stick". For any attack, the service provider can earn at most the total value of all the accounts. Therefore, as long as the collateral required to be put down is larger than this total amount[7], a service provider has no incentive to misbehave, as each misbehavior costs more than what it gains.

6 The Carrot

In this section, we describe how we design the bounty (the carrot in CrudiTEE) to defend against the outsider attacker. The goal is to encourage the outsider attacker to report the wallet signing key breach to the service provider without abusing the signing key.

[7] We believe that a 100% deposit is reasonable because the cost to the service provider is the potential interest they could have earned on the deposit, not the deposit itself.

Throughout this section, we refer to the service provider as the defender, using these two terms interchangeably.

6.1 Desired properties of the Bounty

Distributing signing key shares across multiple TEEs with a threshold signature key generation procedure can lower the chance of signing key breaches caused by outsiders as used in [29]. However, it is not fully resolved. In this section, we further mitigate the risk of unauthorized signatures resulting from side-channel attacks by external attackers with a bounty. The bounty enables the service provider to take appropriate actions before any catastrophic security breaches occur.

The two technical difficulties in the design of the bounty are: (1) how can the attacker and the service provider perform an atomic exchange of the key share and the reward; and (2) how to give the attacker just enough incentive to claim the bounty, while saving the defender's cost. In detail, a good bounty should achieve the following goals:

1. An attacker gets the reward from the service provider if and only if he submits valid proof that convinces the service provider that he has obtained the key share.
2. The construction itself does not leak any knowledge about the key share other than what has already been obtained by the attacker.
3. An attacker prefers submitting the key share(s) to bounty over selling them in the market.
4. An attacker submits the key share as soon as he gets the first key share, instead of continuing the attack.
5. The defender's cost is minimized.

We suggest using smart contract bounty (Section 6.2) to satisfy the goal 1-2. Goals 3-5 are achieved by carefully designing a reward function for submitting key shares for a bounty claim.

6.2 The Smart Contract Bounty

To realize the atomic exchange of the key share and the reward, we initiate the bounty using a smart contract $\mathsf{SC}_{\text{bounty}}$.

As a defense against the outsider attacker, the signing keys are rotated every T units of time. Following each key shares rotation, each TEE computes the hash of all the shares they hold and outputs the hash values to the service provider. The service provider then publishes them in the $\mathsf{SC}_{\text{bounty}}$. The problem arises when the service provider publishes the hash values that do not match the ones generated by the TEEs, making the bounty unable to be claimed. To ensure that the hashes of the key shares are successfully published on the blockchain, we use the proof of publication scheme [14]. In other words, after each rotation or restart, the TEE will verify that the hash of the key shares they are using is the same as the latest version published on the blockchain (via proof of publication). Only then will it use the current key shares to sign the user's requests.

To claim the bounty, the attacker submits the share(s) he finds as proof of knowledge. To prevent front-running, proofs are submitted following a commit-and-reveal scheme [62]. We model this hash function as a random oracle so that it does not leak any information about the key shares themselves.

Upon receiving the key share, the smart contract $\mathsf{SC}_{\text{bounty}}$ checks whether the hash of the share is included in the smart contract. If it is, $\mathsf{SC}_{\text{bounty}}$ puts the reward on hold for a designated period and immediately invalidates all the current secret shares (such that the attacker cannot sell the shares or produce unauthorized signatures after submitting to the bounty). At the same time, the service provider asks the user of the affected accounts to submit insurance in case there exists an unauthorized signature. The attacker gets the reward if there is no insurance claim for the signing key whose shares they are submitting. The amount of the reward is determined by the reward function specified in Section 6.3.

6.3 Reward Function Design

In this subsection, we apply a two-step methodology to the design of the reward function. First, we present a case study focused on the reward function for a deterministic attacker (Section 6.3.2). Then, we broaden the scope to more general scenarios involving non-deterministic attacks (Section 6.3.4 to Section 6.3.7), using observations and insights gained from the simpler case.

6.3.1 Notation and Definition

In this section, we address two types of attackers: the deterministic attacker and the non-deterministic attacker. The deterministic attacker has a fixed deterministic cost function $C(k)$, which is analyzed in Section 6.3.2. The non-deterministic attacker has a fixed cost c_a of attacking one TEE at one step with a certain probability p_s of obtaining one share of the key from the TEE at that step. We deal with them in Section 6.3.4 to Section 6.3.7.

In the smart contract bounty, the reward given to the attacker is determined by a reward function $R(k, t)$, where k is the number of shares that the proof is trying to prove against (i.e., the number of shares obtained by the attacker), and t is the submission time (which is the blockchain timestamp of the inclusion of the bounty-claiming transaction). Essentially, at time t, the attacker provides evidence of having acquired k shares. Since the signing key is rotated every T units of time and the signing key is secret-shared into N shares, we have $t \in [0, T]$ and $k \in [0, N]$.

Recall that we use a (m, N) signature scheme. The service provider has N secrets shares, with $\geq m$ of them together having value v for some $m \leq N$, and $k < m$ of them have value $\mathsf{v} \cdot k/m$. [8] Since m shares are enough to recover the key, the value of m or more shares is the wallet value (i.e. $V(m) = V(m+1) = \cdots = V(N) = \mathsf{v}$).

6.3.2 Case study for deterministic attacker

We first provide a case study with respect to a simpler attacker: he has a deterministic cost function $C(k)$, which is non-decreasing in k, the number of acquired shares.

Naive solution: We start with a naive solution as briefly discussed in Section 1: the linear reward function. In other words, $R(k, t) = V(k) + \eta_1$ for some $\eta_1 > 0$. This is a natural solution: it gives a bit more than how much the share(s) are worth. However, as mentioned, this naive solution can only achieve the goal (3), but not (4) or (5) proposed in Section 6.1. As analyzed, the attacker would continue to attack for more shares and only submit when he has all the key shares.

A starting point: Therefore, we propose first a simple solution that can achieve the goals 3-5 under such a deterministic attack (as the starting point for our real reward function):

$$R(k) = \max_{0<k\leq N}(V(k) - C(k)) + C(1) + \eta_0 + (1 - t/T)\delta_0,$$

[8] Note that in some cases, it may also make sense that having $k < m$ of them has no value. For generality, we consider them to have some partial value.

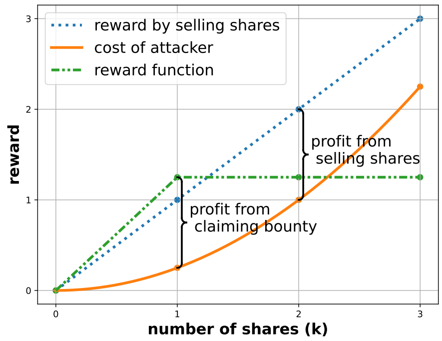


Figure 3 Example of reward function in simplified case.

where η_0 and δ_0 are small constant numbers serving as bonus. This reward function straightforwardly satisfies our goals. For goal (3): Submitting to the bounty provides the attacker with at least η_0 more than selling the shares when the attacker submits with only one share. Consequently, there is no incentive for the attacker to sell the share. For goal (4): Since the adversary achieves maximum profit from the bounty by obtaining just one share $\max_{0\leq k\leq N}(V(k) - C(k)) + \eta_0 + (1 - t/T)\delta_0$, and given that the bonus δ_0 decreases over time, the attacker is incentivized to submit the share to the bounty upon acquiring the first share (and since the adversary needs one share to submit, $C(1)$ is used to compensate this cost). For goal (5): the defender's cost is minimized since the defender cannot spend less. If she reduces her expenditure by η_0, the adversary's gain from the reward might equal the profit from selling the key at point i, where the profit $(V(k) - C(k))$ is maximized. This could lead the attacker to opt for selling the key. As a side property, the attacker also saves cost, as its total cost is always non-decreasing.

A concrete example is depicted in Figure 3. Here, the cost of attack is $C(k) = \frac{1}{4}k^2$, and the value of key shares is $V(k) = k$. The maximum profit for the attacker is $\max_{0\leq k\leq N}(V(k) - C(k)) = V(2) - C(2) = 1$. We set $\eta_0 = \delta_0 = 0.1$. Therefore, the optimal reward function in this scenario is $R(k) = C(1) + (V(2) - C(2)) + \eta_0 = 1.25 + \eta_0$. By structuring the reward function in this way, we not only incentivize the attacker to submit the key share as soon as they get one share but also reduce the defense cost.

Let's compare the reward function we proposed with two baselines: a zero function $R_0(k) = 0$ and a linear reward function $R_l(k) = k + \eta_0$. With R_0, the attacker accumulates 2 shares and sells them in the market, which violates goals 3 and 4. With R_l, the defender pays $2 + \eta_0$ to prevent the attacker from selling 2 shares, which violates the goal 3 and costs more than our reward function.

The main observation from the case study is that giving the attacker more reward at first share is not only a good way to persuade the attacker not to further exploit the key, but also saves the defender's cost.

Of course, here, the context is greatly simplified: the attacker's cost is a known deterministic function of the number of key shares gained. If the attacker's cost is a probabilistic function, the reward function does not always achieve the goals. Also, even for a deterministic attacker with a slightly different cost function, the reward function may not work anymore (e.g., if the attacker costs 10% less per share). Thus, we propose a more complete reward function in Section 6.3.4.

6.3.3 Metrics for Reward Function

While for the deterministic attacker, the simple reward function satisfies all the goals, it becomes more complicated for a non-deterministic attacker, and also when we want to protect against a wider range of attackers. There is a trade-off between goals 3-5 in *Section* 6.1. For example, it would cost more if we wanted to encourage the attacker to turn in the key shares to the bounty earlier. To address this, we turn the goals into numerical metrics and balance them using a weighted average.

We developed three metrics to evaluate how well the reward function meets each of the three specified goals. The first metric is the probability of key shares being sold, denoted as p_e (goal (3)). The second metric is the average holding time, t_h, representing the average time between the attacker finding the first share and the termination of the game (goal (4)). The third metric, the cost to the defender, is denoted as c_d (goal (5)). The cost of the defender is the max between the value the attacker gets by selling the k shares (i.e., $V(k)$) and the amount of the bounty claimed (recall that an attacker can only do one of the two instead of both). To combine these metrics into a score, denoted as f, we introduce parameters α_1 and α_2 to compute a weighted average.

$$f = \alpha_1 \cdot p_e + \alpha_2 \cdot \frac{t_h}{T} + (1 - \alpha_1 - \alpha_2) \cdot \frac{c_d}{\mathsf{v}} \tag{1}$$

In Equation (1), the holding time is normalized by the time period T and the defender's cost is normalized by the value of the key v.

6.3.4 Propose reward function for non-deterministic attacker

We now propose a reward function designed to achieve the objectives outlined in Section 6.1 for a non-deterministic attacker. The optimization and evaluation of this proposed reward function will be detailed in the subsequent parts of this subsection.

To achieve goal (3) in Section 6.1, we need to give more reward to the attacker than the value of the shares. For an attacker with k shares of secret, he can gain $V(k)$ units of money. Thus, to encourage the attacker to submit to the bounty, we give out more than the amount they should have received by selling the key shares. A non-deterministic attacker, however, may get lucky in some cases and get more than one share at a low cost. So our proposed function should have the property $R(k, t) > V(k)$ for all $k \in [1, N]$.

Formally, we give a reward of $V(N)^\epsilon \cdot V(k)^{1-\epsilon} + \eta$ (recall that $dV/dk \geq 0$ for all $k \in [N]$), for some $\epsilon \in [0, 1], \eta > 0$. As long as $\epsilon \geq 0, \eta > 0$, we have $V(N)^\epsilon \cdot V(k)^{1-\epsilon} + \eta > V(k)$ for all $k > 0$. Note that when ϵ increases, we give more reward when $k = 1$, which could potentially reduce the defender's cost (achieving the goal (5)) according to the case study above.

Finally, we need to encourage the adversaries to submit earlier to achieve the goal (4) in Section 6.1. Similarly, we set the "extra bonus" decreasing over time. Formally, let $g(k) := V(N)^\epsilon \cdot V(k)^{1-\epsilon} + \eta - V(k)$ denoting the extra reward we paid to the attacker. We reduce this gain by time: adding a term $-g(k) \cdot t/T$. The reward function we suggest is:

$$R(k, t) := V(N)^\epsilon \cdot V(k)^{1-\epsilon} + \eta - g(k) \cdot t/T, \tag{2}$$

where $g(k) := h(k) + \eta - V(k)$. $\delta \geq 0$, and $\eta > 0$.

To model the real-world constraint of the defender's budget, we also introduce an additional parameter, α_{cap}, into the reward function. This parameter represents the maximum amount of money that the bounty can afford, expressed as a percentage of the secret's value. Specifically, we add a bound $\alpha_{\mathsf{cap}} \cdot V(N)$ to our reward function $R(k,t)$ (Equation (2)), and the resulting new reward function is:

$$\tilde{R}(k,t) = \begin{cases} R(k,t) & \text{if } R(k,t) < \alpha_{\mathsf{cap}} \cdot V(N) \\ \alpha_{\mathsf{cap}} \cdot V(N) & \text{if } R(k,t) \geq \alpha_{\mathsf{cap}} \cdot V(N) \end{cases} \tag{3}$$

where t is the submission time and k is the number of submitted shares ($t \in [0,T], k \in [0,N]$).

6.3.5 Modelling the non-deterministic attacker

To evaluate our function, we first need to model how an attacker behaves. To do this, we first describe the behavior of the attacker that can be modeled as the optimal stopping game. Then, we further find the optimal attacker strategy using a Markov decision process (MDP).

Moreover, with this evaluation result, the defender can quantitatively understand what range of attackers can be effectively prevented using this reward function. She can then change the parameters (e.g., the attacker's ability to begin with and the budget) to modify the function accordingly.

Attacker behavior: We give a detailed description of the attacker's decision process as follows. As in the preceding sections, we exclusively consider a single signing key that is shared among N TEEs. The time period during which the secret remains valid is divided into T discrete time steps. Each time step is further divided into two sub-steps, during which the attacker makes distinct choices: In the first sub-step, the attacker selects the number of TEEs to target during that step. In the second sub-step, the attacker decides whether to terminate the game (sell the shares or claim the bounty) or proceed to the next step. If an attacker decides to target a TEE in a given step, they have a success probability of p_s to acquire a key share from it, while incurring a fixed cost of c_a.

Optimal stopping game: We model an adversary as a player of an "optimal stopping" game [54, 50, 24]. Essentially, the optimal stopping game states the following: there is a sequence of random variables $X_1, X_2, \ldots$ whose distribution is assumed to be known; and there is a sequence of gain functions $(Y_i)_{i\geq 1}$ which take the first i random variables as inputs (i.e., $Y_i(x_1, \ldots, x_i)$ is a function over $x_1 \leftarrow X_1, \ldots, x_i \leftarrow X_i$). Then, the player observes the sequence of random variables one at a time, and for each step i, the player can either stop observing and claim the gain $Y_i(x_1, \ldots, x_i)$ or continue. The goal of the player is to optimize the expected gain. Note that this setting is essentially the same as our setting, where the random variables are the shares gotten by the adversary (e.g. if an attacker can obtain a share with probability p at step i, X_i is a Bernoulli random variable returning 1 with probability p and 0 with probability $1-p$). Then, y_i is the profit the attacker can gain from all the shares he has obtained up to step i, which is the maximum between the value of the bounty and the value of selling these shares, less his cost up to step i. Although some specific forms of optimal stopping games have closed-form solutions (e.g., the secretary problem [19]), for more complex scenarios like ours, a typical approach to find the player's optimal strategy is to model the game with Markov Decision Process (MDP) [54, 50].

MDP: We model the attacking process as an MDP, structuring it into discrete steps. At each step, the attacker decides the number of TEEs to target. The attacker also needs to determine the optimal time to end the attack and obtain their reward: after each step, he must choose to either cease the attack and get the reward or continue attacking in the subsequent step.

We specify the state transition function and the reward function of the MDP as follows. The state of the MDP is defined by the tuple of the number k of shares gained by the attacker, the time slot t, and the sub-step in each time slot $d \in \{0,1\}$. At state $(t,k,0)$, the attacker needs to choose the number of TEEs (denoted as n) to attack in this time slot. The state transitions to $(t, k+\Delta k, 1)$, where Δk is the number of key shares gained in this time slot. The number of newly gained key shares depends on the success rate p_s and the number of TEEs the attacker chooses to attack in that particular step. Specifically, the probability that the attacker gets i new shares in this time slot is $Pr(\Delta k = i) = \binom{n'}{i} p_s^{n'} (1-p_s)^{n'-k}$, where $n' = \max(n, m-k)$. At state $(t,k,1)$, the attacker faces a decision: either end the game by selling the key shares or submitting them to the bounty, or wait until the next time slot. If the attacker chooses to wait until the next time slot, the state will transition to state $(t+1,k,1)$. If the attacker chooses to sell the key shares or submit them to the bounty, the next state will be the termination state. When the time slot reaches the maximum time T at state $(t,T,1)$, the next state will be the termination state.

At each step of the process, the attacker incurs a negative reward of $-c_a \cdot n$, representing the cost of the attacking n TEEs. The attacker gains a positive reward $R(k,t)$ if he submits the key shares to the bounty. Alternatively, if he decides to sell the key shares, he gets $V(k)$. A summary of the transition and reward function of the decision problem is in Table 1.

Table 1 Description of the state transition and reward matrix

State $\times$ Action	State	Probability	Reward
$(k,t,0)\times$attack n TEEs	$(k+i,t,1)$	$Pr(\Delta k = i)$	$-n \cdot c_a$
$(k,t,1)\times$ wait $(t < T)$	$(k,t+1,0)$	1	0
$(k,t,1)\times$ wait $(t = T)$	termination	1	0
$(k,t,1)\times$ turn in	termination	1	$R(k,t)$
$(k,t,1)\times$ selling key	termination	1	$V(k)$

Utilizing the MDP solver [38], we are able to compute the attacker's optimal strategy for a specific reward function. By examining this optimal strategy, we can obtain the metrics defined in Section 6.3.3 (f score). The f score then serves as the objective for optimizing the parameters within the reward function.

6.3.6 Optimize the Reward Function Parameter

In this part, we describe the methodology for deciding the optimal ϵ within the reward function in Equation (2), with α_{cap} as described in Equation (3).

Recall that our reward function $\tilde{R}$ is determined by α_{cap} (bounty cap), ϵ (determining the starting point of the reward), and δ (how fast the reward decays by time). We assume α_{cap} is some constant predefined by the defender, according to her budget.

We now explain our approach for identifying the optimal value of ϵ with regard to the performance metric f. As the defender aims to minimize the cost of the defender, the probability that the attacker will sell the key on the market, and the holding time, the objective is to minimize the score f. When defending against an attacker, the service provider must first decide the parameters used in f (α_1 and α_2) and estimate the ability of the attacker by specifying p_s and c_a. Using the estimated parameters , an optimal ϵ could be numerically computed. Specifically, we discretize $[0,1]$ into a sequence of evenly spaced numbers, calculate a score for each ϵ, and select the one corresponding to the lowest score. [9]

[9] The precision is affected by how many intervals $[0,1]$ is discretized into.

Upon determining the optimal ϵ with estimated parameters, we examine how attackers of various abilities respond to the computed ϵ in the next part. Specifically, these attackers might have different p_s, c_a compared to the initial estimates used for ϵ optimization, representing a range of adversaries stronger or weaker than the initial expectation.

6.3.7 Evaluation Results

We compare the score f of different reward functions, including our reward function, the linear reward function (see below), and no bounty (reward function equals 0).

The linear reward function is a solution that satisfies the goal 3 without considering the cost. Recall that we introduced this naive solution in Section 1 and Section 6.3.2: in the linear reward function,the bounty claimer gets the exact value of share(s) plus a small bonus η_1 to encourage turning in key share(s). We additionally set a time bonus δ_1 that decays with time and encourages early turn-in for the purpose of this case study (to break ties for attacker decisions in MDP), formally given as follows: $R_l(k, t) = V(k) + (1 - t/T)\delta_1 + \eta_1$.

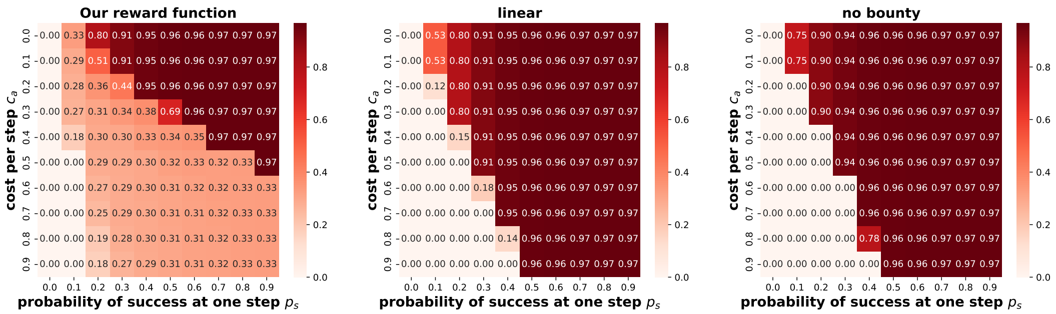


Figure 4 f score for different reward functions. $\alpha_{\mathsf{cap}} = 0.8$. $\alpha_1 = \alpha_2 = 1/3$, $c_a = 0.4$, $p_s = 0.4$, $N = 3$, $\mathsf{v} = 6$. Optimal $\epsilon = 0.95$.

In the evaluation, we set the estimation as $c_a = 0.4$ and $p_s = 0.4$. We set the total number of key shares as $N = 3$ and the value of the key as $\mathsf{v} = 6$, which means the value per share is 2. In expectation, the cost incurred by the attacker to obtain one share is 1 (cost per step / probability of success), resulting in a positive expected profit of 1 for each share acquired. We set $\alpha_1 = \alpha_2 = 1/3$ which means each metric has equal importance. The parameters can be replaced with real-world values when the wallet is implemented in practice. The optimal ϵ we get is 0.95 given the parameters above. Then, we use the optimal parameter to derive the score for attackers with variant cost c_a and success rate p_s.

We show how this function behaves when facing different attackers in Figure 4, where each cell within the heatmap shows the f score corresponding to a specific configuration of the attacker's capabilities, denoted by the parameters c_a and p_s. When the cost is low and the success rate is high (located in the upper right region of the heatmap), the attacker is considered strong. Conversely, when the cost is high and the success rate is low (positioned in the lower left area of the heatmap), the attacker is perceived as weak.

As we can see in the heatmap, when $\alpha_{\mathsf{cap}} = 80\%$, the performance of the reward function we proposed (state of the art) is better than the baseline (no bounty and linear reward function) in most cases. For most attackers, regardless of the ability, our reward function generates a smaller score. The figure demonstrates that our reward function has great performance not just for attackers whose abilities are equal to our estimations ($c_a = 0.4$ and $p_s = 0.4$), but it also works well for stronger attackers. As shown in the figure, essentially for *any* p_s, as long as $c_a \geq 0.4$, the f score is at most 0.3. Similar flexibility on c_a can also be seen in the graph. These results indicate that even without precise attacker ability estimations, our reward function outperforms the alternative reward functions and shows decent effectiveness in preventing outsider attacks.

As mentioned, the defender can then use the heatmap to determine the effectiveness of the reward function given the current attacker's ability estimation and the budget. She may increase her budget to find a reward function that effectively defends against a broader spectrum if needed.

7 Case Study

We briefly discuss how to choose the parameters for the bounty in CrudiTEE using a simple case study. Recall that we need to set time T, the expected return given the number of shares $V(k)$, and the cost function $C(k)$. The calculation below assumes using a (10,20)-threshold signature scheme (i.e., 10 shares are enough to recover the secret) and $T = 30$.

To set the rest of the parameters, we first examine the state-of-the-art side-channel attacks against ECDSA. ECDSA [30] is the most commonly used signature scheme for blockchains like Bitcoin [8], and thus we use it as an example. To our knowledge, all the side-channel attacks without root privilege in recent years against the most popular ECDSA library (OpenSSL [44]) show that they require at least 2^{12} traces to recover a secret [60, 21, 4]. Then, we let the service provider cap the number of signatures a user can make. According to [18], a regular user makes 68 bank transactions per month, which means ~ 2.3 transactions per day. To be lenient, assume the victim makes 230 transactions per day (which is 100x the average number of transactions per day). Since recovering a key share requires at least 2^{12} signatures, which takes ~ 17.8 days. For $V(k)$, recall that we have a rate limit v for each wallet (i.e., the amount of money in each wallet). According to [23], each transaction's average value is 36 dollars for a debit card. We thus set $\mathsf{v} = 36000$, again 100x larger than the average transaction value. Each key share has equal value, and $\mathsf{m} = 10$ shares are enough to recover a key, we set $V(k) = \min(\lceil \mathsf{v} \cdot k/\mathsf{m} \rceil, \mathsf{v})$.

Lastly, we discuss the cost function. The cost function is the most tricky one, since it should capture all the possible costs of an attacker, including operational costs, the risk of being caught, the side channel being mitigated, and so on. Thus, we propose a conservative function (i.e., the minimum cost an attacker can have). Note that for an outsider, the minimum requirement is essentially getting to obtain the traces remotely. The most common way is residing on the same virtual machine as the victim program, as discussed in [48]. Thus, we estimate the cost using the cost of renting the same cloud machine as the service provider. Suppose that it costs $\mathsf{c}_{\mathsf{cloud}}$ dollars per unit of time (e.g., c5.metal from AWS, a commonly used server instance, costs ~\$97.9 per day [6]). Thus, we have $C(k) = \mathsf{c}_{\mathsf{cloud}} \cdot k \cdot 17.8$.

These numbers give us that to recover a key with a value of 36000 dollars, the cost of the attacker is at least ~ 17426 dollars (based on 17.8 days per share, a total of 10 shares, and 97.9 dollars per day for VM). We can come up with a reward function accordingly given all these numbers, along with their budget limit. More accurate numbers can be obtained for a specific service provider by analyzing their own transaction data.

8 Discussion

In this section, we discuss CrudiTEE's performance, limitations and extension application.

Performance Analysis: Reasonable signing performance is required to make the scheme practical. A potential bottleneck of performance may be caused by the secret sharing between different TEEs. In this part, we analyze its concrete performance to show that the multi-TEE ECDSA signing will not be a bottleneck.

For the threshold ECDSA scheme proposed by Gennaro and Goldfeder [22],[10] the benchmark for the signature generation time among m participants is $29 + 24m$ milliseconds. As benchmarked in [40], the highest overhead of TEE is $19.31\times$ in all the tasks tested. Therefore, a conservative signature generation time is around $560 + 463m$ milliseconds. The protocol requires five rounds of communication and we estimate the communication delay for each round as 100 milliseconds [17]. Consequently, the total time for generating a threshold signature is about $1060 + 463m$ milliseconds, which is generally acceptable for cryptocurrency wallets. Additionally, to accommodate high transaction volumes, we can employ multiple sets of TEEs in parallel.

Limitations of insurance: Our techniques provide a technical basis for penalizing the service provider when an attack succeeds against it, providing an incentive for it to properly safeguard its TEEs from outside attackers and a transparent and measurable guarantee to end users. These are significant improvements over the current status quo. Ensuring that the company deposits assets sufficient to satisfy claims against it is a matter for insurance regulators; today, insurance regulators in most jurisdictions require companies to maintain *statutory reserves*, i.e., an amount of cash and readily marketable securities that it can use to pay its foreseeable claims. As with other insurance in real life (e.g., property insurance), users in our system may not be compensated if these reserves (i.e., the company's deposits) are depleted by other claims. Our technical solutions presented here cannot entirely eliminate the need for legal recourse in such situations. Nevertheless, our design provides a stronger foundation for reducing trust in a service provider and for reducing the risk to clients.

Limitations on the type of assets: Note that in most blockchains today, each wallet is tied to a specific private key. Thus, key updates after leakage can cause the assets in the wallet to be non-retrievable. In our paper, we require the asset to be tied to a smart-contract-based wallet, allowing the key updates to work as expected. How to extend our idea to support a wallet without such support remains open.

9 Conclusion

In this paper, we introduced CrudiTEE, a solution designed to mitigate side channels in TEE-based cryptocurrency wallets by leveraging economic incentives. Our wallet authentication system utilizes OAuth to ensure both accountability and user-friendliness. Additionally, we designed a combination of stick (insurance) and carrot (bounty) to safeguard against both insider and outsider attacks. Finally, we evaluated our approach and showed its effectiveness.

References

1 O. Aciiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. In *Topics in Cryptology – CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, pages 271–286, February 2007.

2 AMD secure encrypted virtualization (SEV). `https://www.amd.com/en/developer/sev.html`.

[10] This scheme considers malicious participants, so there are unnecessary steps in the protocol if we assume all the participants are honest, which is true in our case.

3 Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

4 Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. Ladderleak: Breaking ecdsa with less than one bit of nonce leakage. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 225–242, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3372297.3417268`.

5 The sequencer and censorship resistance. URL: `https://docs.arbitrum.io/sequencer/#unhappyuncommon-case-sequencer-isnt-doing-its-job`.

6 AWS price calculator. `https://calculator.aws/`, 2023.

7 Kushal Babel, Nerla Jean-Louis, Mahimna Kelkar, Yunqi Li, Carolina Ortega Perez, Aditya Asgoankar, Sylvain Bellemare, Ari Juels, and Andrew Miller. The Sting framework (SF), 2023. URL: `https://initc3org.medium.com/the-sting-framework-sf-ef00702c88c7`.

8 Bitcoin core 25.0. `https://github.com/bitcoin/bitcoin`, 2023.

9 Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. Dr. SGX: Automated and adjustable side-channel protection for SGX using data location randomization. In *35th Annual Computer Security Applications Conference*, pages 788–800, 2019.

10 Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In *27th USENIX Security Symposium*, pages 1335–1352, 2018.

11 Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. Mitigating cache/timing attacks in AES and RSA software implementations. In *RSA Conference*, 2006.

12 D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

13 Iadine Chadès, Guillaume Chapron, Marie-Josée Cros, Frédérick Garcia, and Régis Sabbadin. Mdptoolbox: a multi-platform toolbox to solve stochastic dynamic programming problems. *Ecography*, 37(9):916–920, 2014.

14 Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.

15 Coinbase. `https://www.coinbase.com/`.

16 Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. The pyramid scheme: Oblivious RAM for trusted processors. *arXiv preprint arXiv:1712.07882*, 2017.

17 Luca De Vito, Sergio Rapuano, and Laura Tomaciello. One-way delay measurement: State of the art. *IEEE Transactions on Instrumentation and Measurement*, 57(12):2742–2750, 2008.

18 Federal Reserve Bank of Atlanta. Survey of consumer payment choice 2020, 2020. `https://www.atlantafed.org/-/media/documents/banking/consumer-payments/survey-of-consumer-payment-choice/2020/2020-survey-of-consumer-payment-choice.pdf`.

19 Thomas S. Ferguson. Who Solved the Secretary Problem? *Statistical Science*, 4(3):282 – 289, 1989. `doi:10.1214/ss/1177012493`.

20 K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261, May 2001.

21 Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. Ecdsa key extraction from mobile devices via nonintrusive physical side channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS

'16, page 1626–1638, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978353.

22 Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1179–1194, 2018.

23 Geoffrey Gerdes, Claire Greene, Xuemei (May) Liu, Emily Massaro, Ambika Nair, Zach Proom, Nancy Donahue, Lisa Gillispie, Mary Kepler, Doug King, Susan Krupkowski, Ellen Levy, Dave Lott, Mark Manuszak, David Mills, Laura Reiter, Stephanie Sciuletti, Susan Stawick, Catherine Thaliath, Jessica Washington, and Julius Weyman. The 2019 federal reserve payments study. https://www.federalreserve.gov/paymentsystems/2019-December-The-Federal-Reserve-Payments-Study.htm.

24 Alexander V. Gnedin and Ulrich Krengel. A stochastic game of optimal stopping and order selection. *Annals of Applied Probability*, 5:310–321, 1995. URL: https://api.semanticscholar.org/CorpusID:122457776.

25 Google LLC. Using OAuth2.0 with OpenID Connect in Google. https://developers.google.com/identity/openid-connect/openid-connect.

26 Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *DIMVA 2016: Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 9721 of *Lecture Notes in Computer Science*, pages 279–299, 2016.

27 D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security & Privacy*, pages 490–505, 2011.

28 H100 tensor core GPU | NVIDIA. https://www.nvidia.com/en-us/data-center/h100/.

29 Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Advances in Cryptology – CRYPTO '95*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352, 1995.

30 Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). In *International Journal of Information Security*. Association for Computing Machinery, July 2001. URL: https://doi.org/10.1007/s102070100002.

31 P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397, August 1999.

32 P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, 1996.

33 Jiasun Li. On the security of optimistic blockchain mechanisms. *Available at SSRN 4499357*, 2023.

34 Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In *30th USENIX Security Symposium*, August 2021.

35 Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 355–371, 2021. doi:10.1109/SP40001.2021.00063.

36 Suresh S Malladi and Hemang C Subramanian. Bug bounty programs for cybersecurity: Practices, issues, and recommendations. *IEEE Software*, 37(1):31–39, 2019.

37 Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, page 1, New York, NY, USA, June 2013. Association for Computing Machinery. doi:10.1145/2487726.2488368.

38 Markov decision process (mdp) toolbox. https://pymdptoolbox.readthedocs.io/en/latest/api/mdptoolbox.html.

39 ID tokens in the Microsoft identity platform. https://learn.microsoft.com/en-us/azure/active-directory/develop/id-tokens.

40 Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. A comparison study of intel sgx and amd memory encryption technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2018.

41 S.-J. Moon, V. Sekar, and M. K. Reiter. Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration. In *22nd ACM Conference on Computer and Communications Security*, pages 1595–1606, October 2015.

42 M. Morbitzer, S. Proskurin, M. Radev, M. Dorfhuber, and E. Salas. Severity: Code injection attacks against encrypted virtual machines. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 444–455, Los Alamitos, CA, USA, may 2021. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/SPW53761.2021.00063, doi:10.1109/SPW53761.2021.00063.

43 Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on Intel SGX. *arXiv preprint arXiv:2006.13598*, 2020.

44 OpenSSL. https://www.openssl.org/, 2023.

45 D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20, 2006.

46 Aaron Parecki. OAuth 2.0 basic information. https://developers.google.com/identity/openid-connect/openid-connect.

47 J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and countermeasures for smart cards. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210, September 2001.

48 Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 199–212, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1653662.1653687.

49 Carlton Shepherd, Konstantinos Markantonakis, Nico van Heijningen, Driss Aboulkassimi, Clément Gaine, Thibaut Heckmann, and David Naccache. Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis. *Computers & Security*, 111:102471, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0167404821002959, doi:10.1016/j.cose.2021.102471.

50 Albert N. Shiryaev. *Optimal Stopping Rules*, pages 1032–1034. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-04898-2_433.

51 Saurabh Suratkar, Mahesh Shirole, and Sunil Bhirud. Cryptocurrency wallet: A review. In *2020 4th international conference on computer, communication and signal processing (ICCCSP)*, pages 1–7. IEEE, 2020.

52 J. Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 3:219–234, September 2019.

53 Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 19–34, April 2017. doi:10.1109/EuroSP.2017.28.

54 J.N. Tsitsiklis and B. van Roy. Optimal stopping of markov processes: Hilbert space theory, approximation algorithms, and an application to pricing high-dimensional financial derivatives. *IEEE Transactions on Automatic Control*, 44(10):1840–1851, 1999. doi:10.1109/9.793723.

55 Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxeattack.com/, 2020.

56 Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium*, August 2015.

57 Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *24th ACM Conference on Computer and Communications Security*, October 2017.

58 M. Weiß, B. Heinz, and F. Stumpf. A cache timing attack on AES in virtualization environments. In *16th International Conference on Financial Cryptography and Data Security*, February 2012.

59 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

60 Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. Cryptology ePrint Archive, Paper 2014/140, 2014. `https://eprint.iacr.org/2014/140`. URL: `https://eprint.iacr.org/2014/140`.

61 Martin Young. Coinbase custodies 11% of entire crypto capitalization. URL: `https://cointelegraph.com/news/coinbase-custodies-11-of-entire-crypto-capitalization`.

62 Zainan Victor Zhou and Matt Stam. Rc-5732: Commit interface: A simple but general commit interface to support commit-reveal scheme. `https://eips.ethereum.org/EIPS/eip-5732`, September 2022.

63 Dionysis Zindros. Hours of Horus: Keyless cryptocurrency wallets. *Cryptology ePrint Archive*, 2021.