The Forking Way: When TEEs Meet Consensus

Annika Wilde
Ruhr University Bochum
annika.wilde@rub.de

Tim Niklas Gruel
Ruhr University Bochum
tim.gruel@rub.de

Claudio Soriente
NEC Laboratories Europe
claudio.soriente@neclab.eu

Ghassan Karame
Ruhr University Bochum
ghassan@karame.org

***Abstract*—An increasing number of distributed platforms combine Trusted Execution Environments (TEEs) with blockchains. Indeed, many hail the combination of TEEs and blockchains a good "marriage": TEEs bring confidential computing to the blockchain while the consensus layer could help defend TEEs from forking attacks.**

In this paper, we systemize how current blockchain solutions integrate TEEs and to what extent they are secure against forking attacks. To do so, we thoroughly analyze 29 proposals for TEE-based blockchains, ranging from academic proposals to production-ready platforms. We uncover a lack of consensus in the community on how to combine TEEs and blockchains. In particular, we identify four broad means to interconnect TEEs with consensus, analyze their limitations, and discuss possible remedies. Our analysis also reveals previously undocumented forking attacks on three production-ready TEE-based blockchains: Ten, Phala, and the Secret Network. We leverage our analysis to propose effective countermeasures against those vulnerabilities; we responsibly disclosed our findings to the developers of each affected platform.

I. Introduction

Modern blockchains leverage smart contracts to run arbitrary business logic. Smart contracts instantiate state machine replication where all miners are expected to execute the contract code over common inputs and update their state. Here, the (binary) code of the smart contract and the transaction data must be available to all miners. In the case of applications handling sensitive data, this can hardly be tolerated.

Researchers and practitioners tried to address this gap by protecting the contract logic and the corresponding transaction data/state. Available solutions rely on trusted third parties to execute the contracts [1], [2], zk-rollups [3]–[5], secure multi-party computation (MPC) [6], [7], or Trusted Execution Environments (TEEs) [8]–[11]. TEE-based solutions emerge as an attractive means to ensure the confidentiality of the contract and the associated data. Namely, they are (1) more efficient and more expressive compared to solutions based on MPC and (2) require drastically lower deployment costs compared to solutions that require trusted third parties. These factors led to considerable adoption of TEEs within existing decentralized platforms, such as Ten [12], Phala [9], and the Secret Network [13]. Given the pervasiveness of TEEs nowadays, the number of decentralized platforms supporting TEEs is only expected to grow.

While TEEs bring several benefits to blockchains (e.g., confidential computing for smart contracts), they can also leverage the consistency layer of the underlying blockchain to mitigate one of their fundamental limitations: the lack of proper countermeasures against so-called *forking attacks* [14]. Such attacks can be mitigated if the TEE processes requests that are properly serialized. Consensus protocols, in general, and blockchains, in particular, are notorious for ensuring the total ordering of events. Hence, TEE-based applications can naturally rely on blockchains to counter forking attacks [15], [16]. On an abstract level, such a "marriage" between blockchains and TEEs supports the confidential execution of contract logic while mitigating forking attacks on TEEs.

In this paper, we provide a systematization of existing solutions used by various TEE-based blockchains to counter forking attacks against the enclave. We point out that a forking attack against TEEs can be carried out either by rolling back its state or by cloning the TEE instance [14]. Unfortunately, while previous work [15], [16] investigated forking attacks based on rollbacks, it did not consider practical forking threats that can arise from cloning. In other words, a TEE-based blockchain that includes anti-rollback mechanisms may still be susceptible to forking attacks based on cloning. Towards this end, we analyze 29 proposals for TEE-based blockchains, ranging from academic proposals to production-ready platforms. We identify four broad categories of anti-forking techniques used in TEE-based blockchains. Further, we analyze the trade-offs of each technique—ranging from the expressiveness of the smart contract that can be deployed in the TEE to the restriction on the L1 layer that can be used. We then highlight pitfalls in how these techniques are currently instantiated and discuss workable mechanisms to practically address those pitfalls. Our analysis shows that combining TEEs and blockchains to provide forking-resistant confidential smart contracts presents a number of practical challenges that are often overlooked by researchers and practitioners. In particular, we show that (1) stateless enclaves can be protected against forking attacks in existing protocols by leveraging ephemeral enclave identities, but (2) devising comprehensive solutions to protecting stateful enclaves in existing TEE-based blockchains depends on several factors, such as the type of consensus (final or eventual) or the throughput of the consensus layer, among others.

Throughout our systematic analysis, we identify several vulnerabilities that lead to forks in the TEE state. Among these

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241934
www.ndss-symposium.org

vulnerabilities, we describe previously undocumented forking attacks on three production-ready TEE-based blockchains, namely: Ten [12], Phala [9], and the Secret Network [13].

In summary, we make the following contributions:

Systemization of knowledge: We provide a systemization of existing solutions used to counter forking attacks on TEE-based blockchains. We analyze, by means of a thorough empirical analysis, 29 proposals for TEE-based blockchains. We categorize existing anti-forking mechanisms used by such platforms into four broad categories and identify their gaps and limitations (cf. Section IV).

Pitfalls in TEE-based blockchains: We explore the solution space to secure TEE-based blockchains against possible forking attacks. Namely, we discuss the various trade-offs existing solutions exhibit and show significant pitfalls in how these techniques are instantiated. For example, Li *et al.* [16] note that FastKitten [8] cannot be forked by rollback. However, we show a new cloning-based forking attack against FastKitten (cf. Section IV-C). We discuss workable mechanisms to address those pitfalls practically and show that the underlying choice of the mitigation technique depends mostly on the provisions of Layer 1.

Cloning vulnerabilities in production-ready networks: We present and evaluate previously undocumented cloning vulnerabilities in Ten, Phala, and the Secret Network, three production-ready TEE-based blockchains (cf. Section V). Our first attack against Phala enables an adversary to operate two instances of the enclave contract (with the same code and address) and freely choose the instance to answer a request; given that the two instances share the code and the address, a client cannot distinguish which one answered its request. This allows the adversary to reply with a stale state and works despite defenses like transaction ordering and other well-known measures used to prevent rogue contract injections [17]. Similarly, in our second attack against the Secret Network, the adversary uses another instance of the same contract code to answer the query incorrectly based on a different state. This new attack works despite the anti-rollback measures proposed by Jean-Louis *et al.* [15]. Our final novel attack against Ten allows an adversary to spawn as many enclaves to increase the chances that it is elected as the next rollup proposer despite the anti-rollback solution used in Ten.

Practical countermeasures: By leveraging our systematic analysis, we discuss and analyze practical and workable solutions, based on our findings, to address the vulnerabilities identified in Ten, Phala, and the Secret Network.

Responsible disclosure: We responsibly disclosed our findings and suggested countermeasures to the developers of these production-ready TEE-based blockchains, respectively (see https://cloning-tee-blockchains.github.io/).

II. Background and Related Work

A. Hardware-based TEEs

Trusted Execution Environments leverage the hardware to control access to runtime memory by software, thereby providing an isolated sandbox—known as "enclave"—to execute user code. As such, the threat model for TEEs includes malicious user (peer) processes and a malicious OS while the underlying hardware is trusted. Commercial TEEs include Intel SGX [18], AMD SEV [19], or ARM TrustZone [20]. While each of those commercial TEEs has its own unique features, they all share a common blueprint. In the following, we will only discuss the TEE features that are relevant for this work and refer the reader to [21] for a complete treatment of TEEs.

Attestation and Enclave Identity. Attestation allows (remote) verifiers to check the code that is running within an enclave and the configuration of the underlying platform. In a nutshell, a trusted system component uses a private key to sign a hash computed over the code deployed in the enclave and various attributes of the machine (e.g., TEE version, security patches); the verifier uses a public key, usually distributed by the TEE manufacturer, to verify the signature.

The hash computed over the code and the machine attributes is often referred to as the "identity" of the enclave and allows to distinguish two enclaves running two different binaries on the same machine or two enclaves running the same binary but on two different machines. Note that TEEs provide no support to distinguish enclaves with the same binary on the same platform.

Sealing. Apart from secure runtime memory, TEEs also provide secure (disk) storage. This is achieved by means of authenticated encryption and by leveraging so-called *sealing* keys. A sealing key for a given enclave is derived from a platform-specific master key and the identity of the enclave. Hence, two enclaves running different binaries (or on different platforms) cannot access the same sealing key; as a result, data sealed by one enclave cannot be unsealed by the other. Nevertheless, two enclaves on the same platform running the same binary have access to the same sealing key. We point out that access to the disk is mediated by a possibly malicious OS; hence, sealing provides no freshness guarantees. That is, when the enclave fetches a sealed state from the disk, it has no means to distinguish whether the ciphertext provided by the OS corresponds to (1) the latest sealed state or (2) to an older ciphertext containing a stale state.

B. Forking Attacks on TEEs

Forking attacks are well-known threats to the consistency of distributed applications [22]. In the context of TEEs, a forking attack leverages the lack of freshness of the sealing functionality or the lack of mechanisms to distinguish two instances of the same enclave application. In other words, a forking attack on a TEE can be mounted either by rolling back the enclave state or cloning the enclave application.

In the following, we describe both strategies with a sample TEE application denoted as E. Consider the case where E

updates its state s_j based on a function f, the previous state s_{j-1}, and an input i. Let s_0 be the enclave starting state, and assume that E seals its state to disk after every update to recover it in case of crashes. Furthermore, let E receive two inputs, i_1 and i_2, one after the other. In a benign setting, the enclave will move through states $s_1 = f(s_0, i_1)$ and $s_2 = f(s_1, i_2)$, each sealed to disk.

Rollback-based forking. Forking can be achieved by terminating the enclave after it has sealed s_1. Next, E is restarted; when the enclave fetches state from disk, the adversary provides s_0 instead of s_1, thereby rolling back the state of the enclave. If the enclave then processes i_2, it moves to state $s'_2 = f(s_0, i_2)$.

Cloning-based forking. Here, the adversary launches another enclave instance running the same code—denoted as E'—on the same machine. Note that, as discussed above, it is not possible to distinguish the two instances apart. Hence, the adversary feeds i_1 to E that advances to state $s_1 = f(s_0, i_1)$ and feeds i_2 to E' that advances to state $s'_1 = f(s_0, i_2)$.

C. Layer One, Layer Two, and Blockchain Applications

A blockchain is a decentralized system that ensures state consistency among the system's nodes. Consistency is maintained through a "Layer One" (L1) consensus protocol, which ensures that the majority of (honest) nodes agree on a common state. Transactions are used to update the state, and nodes typically use the consensus protocol to agree on the order of transactions, organized in batches called blocks. For instance, Nakamoto-style blockchains (such as Bitcoin and Ethereum Classic) use a randomized leader election protocol (leveraging Proof of Work or Proof of Stake) to elect the next block proposer but rely on the longest chain rule to reach consensus on transactions and blocks. Such protocols scale well to a high number of nodes but only achieve modest throughput.

Layer Two (L2) solutions overcome the performance and functionality limitations of L1. For example, payment channels [23] and roll-ups [24] enable the bulk processing of transactions, thereby decreasing the transaction processing load on L1. Similarly, cross-chain bridges [25] and atomic swap protocols [26] enable interaction between independent blockchains. These solutions, however, still rely on the underlying L1 blockchain for final confirmation and validation. For example, L2 solutions that process batches of transactions off-chain must periodically commit state updates to the L1 blockchain, which provides final confirmation for those off-chain transactions.

Some blockchain applications interface L1 blockchains with L2 solutions. For example, lightweight clients [27] monitor the blockchain by storing only block headers, querying transactions from full nodes, and verifying the responses using these headers. Other applications like wallets [28] securely manage user accounts and the keys necessary to authorize transactions.

D. Related Work

To the best of our knowledge, no previous work has focused on forking attacks against TEE-based blockchains. We now discuss the most relevant related papers.

Li *et al.* [16] systematize TEE-assisted confidential smart contracts with respect to "Privacy-Preserving Properties" (e.g., I/O privacy) and "Blockchain Intrinsic Benefits" (e.g., high availability). The authors of [16] briefly mention "state consistency" and the problem of state freshness for TEEs. Nevertheless, Li *et al.* [16] do not provide a systematic study on forking vulnerabilities of TEE-based blockchains. Jean-Louis *et al.* [15] investigate privacy flaws in four production-ready blockchains, namely Phala, Ten, Oasis, and the Secret Network. The authors discuss the privacy leaks due to a potentially malicious OS that sees enclave accesses to an external (encrypted) database or the enclave page faults. The authors of [15] also show a rollback attack on the Secret Network that allows the adversary to learn private information about transactions (e.g., transaction amounts, balances, etc.). However, Jean-Louis *et al.* [15] provide no attacks on the other platforms they consider, nor do they take into account attacks based on cloning the enclave.

III. Systemization Methodology

In what follows, we explain our classification criteria and outline our systemization methodology. Figure 1 provides an overview of our approach.

A. Selection Criteria

To the best of our knowledge, the majority of TEE-based blockchains rely on Intel SGX, with two notable exceptions: CCF [29], which supports both Intel SGX and AMD SEV, and TZ4Fabric [30], which is based on ARM TrustZone. Consequently, our systematic evaluation focused solely on SGX-based blockchains. However, we believe our findings are applicable to other TEEs, as they share similar limitations.

The selected platforms were taken from a curated list of SGX-based blockchains [31, Sec. Blockchains/Session 4] and an SoK on TEE-assisted confidential smart contracts [16], totaling 41 platforms. We discarded platforms without an enclave instantiation, platforms that were archived, or that had no English documentation, ending up with 28 platforms. We considered an additional production-ready platform, Ten [12], that was studied by [15]—one of the closest related works.

B. System Classification

We systemize the platforms in our analysis based on how they leverage functionality from TEEs. We identify four main categories:

Category 1—TEE-based Smart Contracts: Blockchains in this category use the TEE to achieve confidentiality in the execution of the smart contract. That is, the TEE fetches encrypted inputs from L1, processes the transactions, and pushes encrypted outputs to L1. Some blockchains [29], [32] employ state-machine replication, where the smart contract state is replicated across multiple enclaves, whereas others [9]

assign a single enclave per contract and verify its integrity via attestation. The contract's state may be stored locally within the enclave [13], [29], [32], or on the blockchain itself [11]. To prevent a malicious smart contract from accessing the state of another (a scenario known as rogue smart contract code injection [17]), the enclave typically binds the contract address (derived from the code hash) with the state. Note that the blockchain may allow clients to issue *contract queries*, i.e., direct read requests to the smart contract that bypass the ledger and allow the client to obtain information on the contract state [9], [13].

Category 2—TEE-based Consensus Protocols: Blockchains in this category use the TEE to speed up or scale consensus. Some blockchains [33]–[35] leverage the TEE for secure leader election. For example, the "Proof of Luck" [34] consensus protocol uses the TEE as a source of unbiased randomness to select the next block proposer. Other systems [29], [36] execute the consensus mechanism directly within the TEE to improve scalability. Since the code inside the TEE is assumed to be trusted after attestation, this approach dramatically reduces communication complexity in the presence of malicious nodes.

Category 3—TEE-based L2 Solutions: Blockchains in this category leverage the TEE to implement confidential L2 solutions. Some blockchains [8], [10], [37], [38] use the TEE to instantiate confidential smart contracts for L1 blockchains that lack native support for such features. Other L2 solutions focus on supporting confidential operations over transactions, including mixers [39], payment channels [40]–[42], and cross-chain bridges [43].

Category 4—TEE-based Blockchain Applications: Last but not least, some applications leverage TEEs to enable secure access to the blockchain. This includes the secure storage of key material required for blockchain interaction [44], [45], secure fetching and validation of blocks [1], and secure data retrieval for blockchain-based applications [46].

C. Methodology

We analyze the selected platforms to identify the various techniques used to prevent forking attacks. We detect four broad techniques. Some platforms use *stateless enclaves* that cannot be rolled back "by design" but may be vulnerable to cloning. Other platforms rely on *ephemeral enclave IDs* to distinguish clones. Another technique is to rely on a *fixed set of clients* that monitors the enclave to detect forks. Finally, a common alternative anti-forking technique is to *serialize the enclave state* by using the ledger.

These techniques, however, incur several trade-offs in terms of *functionality*, *robustness*, and *performance*. For example, stateless enclaves cannot run stateful applications, thereby restricting the functionality offered by the smart contract. Similarly, using ephemeral IDs or a fixed set of clients hinders identity management and complicates the addition/removal of nodes. Finally, using the ledger to serialize the enclave state bounds the enclave throughput to the throughput of the ledger, thereby reducing performance.

In the following sections, we systematize the selected 29 platforms based on the anti-forking mechanisms they use, assess their resistance to forking attacks, and analyze their robustness, performance, and functionality trade-offs.

IV. Pitfalls in TEE-based Blockchains

We assume the typical threat model for SGX applications, where the hardware is part of the trusted computing base (TCB), but any privileged software, such as the operating system (OS), is considered potentially malicious. In this model, the adversary fully controls system resources, including memory, storage, and network communication, but cannot compromise the hardware. Our analysis focuses explicitly on forking attacks within the context of TEE-based blockchains. As such, we assume that the enclave remains uncompromised and that side-channel and denial-of-service (DoS) attacks are out of scope. Whenever applicable, we assume that the adversary controls a node running the enclave. Since the adversary controls the communication between the enclave and the rest of the system, they can drop or modify all inputs and outputs of the enclave. For instance, the adversary can provide the enclave with a stale state whenever the enclave fetches the state from the disk or the blockchain. Further, the adversary can clone the enclave by launching an arbitrary number of instances of the same TEE binary. As mentioned in Section II, SGX provides no means to distinguish two enclaves running the same binary on the same platform, and the two binaries can access the same sealed state.

A. Stateless Enclaves

Overview. A stateless enclave produces output depending only on the current input and does not need to maintain the state of previous computations. Hence, if restarted, the enclave fetches no state from persistent storage [9], [13], [33]–[35], [38], [39], [41]–[44], [46]–[50]. At times, the enclave may use an immutable state, such as a signing key, fetched from persistent storage upon every restart. However, the state (i.e., the signing key) never changes. Prominent examples of stateless enclaves are transaction mixers [39] that output a permutation of the set of transactions received as input. Differently, an enclave that implements a standard database is typically not stateless—since previous queries may determine the result of the current query—and the database is periodically sealed to disk to be fetched upon restarts.

Vulnerability to Forking Attacks: Stateless enclaves are resistant to rollback attacks *by design*. If the enclave fetches no state from persistent storage or fetches an immutable state, then the adversary has no means to roll back the enclave. Nevertheless, cloning attacks remain viable in this setting. For example, if the computation is randomized, the adversary can launch multiple clones and select the more favorable output.

Limitation 1—Expressiveness. A stateless enclave clearly limits the type of applications that can be deployed in the

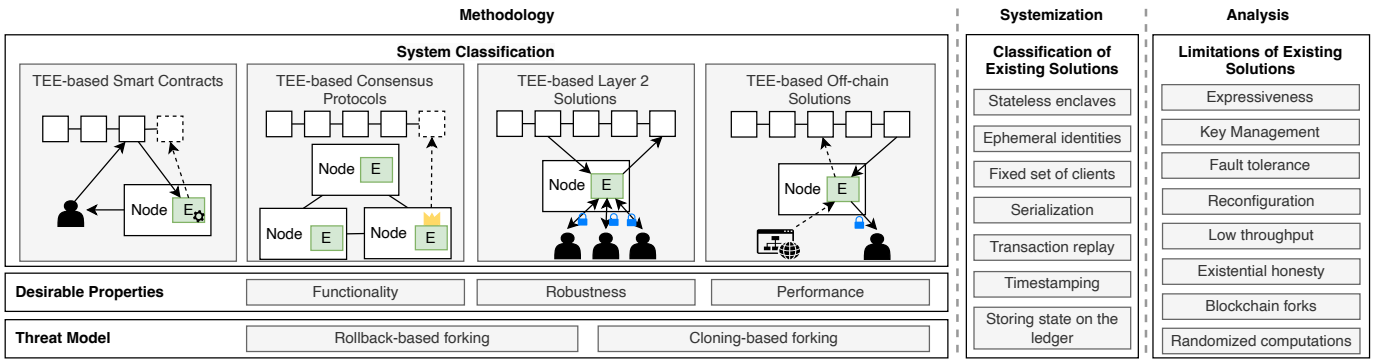

Fig. 1. Overview of our systemization methodology: We classify SGX-based blockchains into four distinct categories and analyze their tradeoffs with respect to robustness, functionality, and performance.

smart contract. For example, a database application requires enclaves to keep a persistent state and cannot be deployed in a smart contract running within a stateless enclave.

Example: PoUW. Zhang *et al.* [33] propose a leader election protocol based on Proof-of-Useful-Work (PoUW). PoUW is an alternative to well-known Proof-of-Work protocols where miners "waste" resources to solve a cryptographic puzzle. In PoUW, mining resources are used to carry out useful tasks submitted by clients. A miner can propose a new block only if it produces "proof" that it has completed a specific task. In particular, after completion of a task, the miner draws a random number $r \in [0, 1]$ and proposes a new block only if r is greater than a threshold t set to $1 - (1 - \text{diff})^n$, where diff is a tuneable difficulty parameter, and n is the number of instructions required to complete the task—so that tasks with more instructions increase the chances of the miner to propose a block. The proof is, essentially, the task output out and the random number r. PoUW leverages TEEs to ensure that the miner faithfully completes the task and draws an unbiased random r. In particular, the task is executed within an enclave; upon completing the task, the enclave draws r and signs the proof with its signing key. Other nodes can verify the proof using the corresponding public key. Remote attestation allows nodes to check that the miner indeed runs a legitimate PoUW enclave. Note that enclave signatures also include the current block's hash so that a PoUW has limited validity. This measure prevents rollback attacks: a malicious host may feed (the hash of) a stale block to its PoUW enclave, but the proof that the enclave outputs will not be accepted by other nodes—since it is not tied to the current block.

Despite being secure against rollback attacks, a cloning attack on PoUW is still possible, as shown in Figure 2. Assume a malicious miner that receives a *task* from a client (step ①). The miner starts two PoUW enclaves E_{PoUW} and E'_{PoUW} and provides them with the received task, the current block, and difficulty (step ②). The enclaves execute the task, yielding the same out and n, so they compute the same threshold t. E_{PoUW} and E'_{PoUW} now draw random numbers r and r', respectively. Assume that $r' > t$ while $r \leq t$ (step ③).

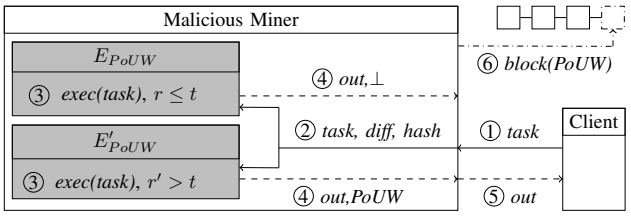

Fig. 2. Example of a cloning attack on PoUW [33]. A malicious miner starts two PoUW enclaves to increase its chances that one of its enclaves produces proof of useful work.

Consequently, E'_{PoUW} returns a PoUW, while E_{PoUW} does not (step ④). The adversary returns the output to the client (step ⑤) and proposes a new block with the PoUW generated by E'_{PoUW} (step ⑥). The adversary effectively increased its chances of proposing the next block by running two clones of the PoUW enclave. The adversary may run > 2 PoUW enclaves to increase its chances of proposing the next block further.

Takeaway 1 – Stateless Enclaves. Using enclaves that do not keep a persistent state protects against rollback attacks by design. However, stateless enclaves limit the expressiveness of the TEE application and do not deter cloning attacks when the TEE application is non-deterministic.

B. Ephemeral Identities

Overview. Most TEE-based blockchains use enclaves with long-lasting identities [12], [35], [37], [40], [46], [49]. For example, one can identify an enclave with its public key, and the key pair is sealed to disk to recover from crashes. However, some blockchains assign ephemeral IDs to enclaves [35], [39], [41]–[43], [45], [51]. That is, the enclave generates a fresh key pair at runtime, and the public key is used as an identifier. The key pair is not sealed to the disk. Hence, if the enclave crashes and is restarted, it obtains a new identity.

Vulnerability to Forking Attacks: This simple technique prevents cloning attacks since ephemeral public keys allow external parties to distinguish two enclave instances, even if they run the same binary on the same platform—since they will likely generate two different key pairs. Hence, messages encrypted under the ephemeral public key of one enclave instance cannot be decrypted by any other instance. Note that if the enclave seals the state to disk, this solution does not protect against rollback attacks. We note that all the applications we analyzed and used ephemeral keys have stateless enclaves.

Limitation 2—Key management. Ephemeral keys as an anti-cloning mechanism may require proper key management. That is, the consensus protocol must keep track of participating enclaves and their (ephemeral) public keys. For example, in the case of PoUW [33], an ephemeral key per enclave would prevent the adversary from using several instances of the PoUW enclave as long as there is a consistent layer that keeps track of all registered ephemeral public keys. This requires a dedicated registration process. Further, when an enclave crashes, a mechanism is needed to remove the old ephemeral key from the list of participating enclaves and add the new one (created by the enclave upon restart).

In case the application logic can tolerate multiple clones of an enclave without causing harm, there is no need to rely on a dedicated key management mechanism. For instance, Tesseract [43] allows clients to send time-locked deposits by encrypting coins under the ephemeral public key of the Tesseract enclave. The enclave does not seal state, and, in case of a crash, coins are automatically reverted to the original client account after the time-lock expires. Thus, the Tesseract enclave logic is inherently robust to cloning attacks; that is, an adversary gains no advantage by cloning the Tesseract enclave.

Example: Twilight. Dotan *et al.* [41] propose Twilight, a differentially private payment channel network. Payment channels are established between two relays, and assets can be transferred between them, bypassing the blockchain. Two parties that are not connected via a payment channel can transfer coins using a sequence of hops between relays. At startup, the Twilight enclave generates an ephemeral key pair. Other relays use the public key of a relay to send encrypted payment information. Note that Twilight enclaves do not store state on disk. Thus, even if they cannot tolerate crashes, they resist rollback attacks by design (cf. Section IV-A).

Figure 3 shows how Twilight leverages ephemeral keys to prevent cloning attacks. Assume an honest relay R_H and a malicious relay R_M. R_H operates Twilight enclave E_H, whereas R_M operates two enclaves E_M and E'_M. Each enclave generates a key pair. E_H and E_M exchanged public keys to establish a payment channel. A dedicated smart contract binds the payment channel to the ephemeral keys of E_H and E_M (addressing Limitation 2). Assume that E_H wants to send payment p over this channel and encrypts it with pk_{E_M}, resulting in ciphertext c (step ①). E_H outputs c to R_H which sends it to R_M (step ②). R_M tries to claim the payment twice, sending it to both enclave instances (step ③). E_M decrypts

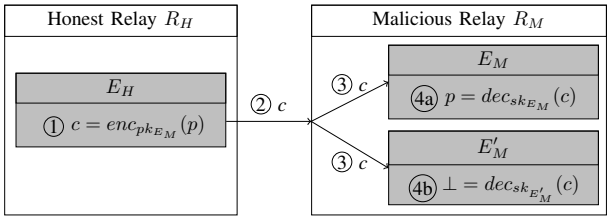

Fig. 3. Example of a (failed) cloning attack on Twilight [41]. A malicious relay forwards an encrypted payment to two instances of the Twilight enclave. Ephemeral keys prevent enclave E'_M from decrypting the ciphertext.

c and retrieves p because it has the secret key corresponding to pk_{E_M} (step 4a). However, E'_M has a different secret key; hence decryption of c fails (step 4b).

> **Takeaway 2 – Ephemeral Identities.** Identifying each enclave by means of an ephemeral ID (i.e., renewed at restart) can prevent cloning attacks. In settings where the state needs to persist, one should additionally rely on anti-rollback mechanisms.

C. Fixed Set of Clients

Overview: Some TEE-based blockchains only allow a fixed set of clients to interact with the smart contract [8], [40]. For example, FastKitten [8] implements a smart-contract solution on top of Bitcoin. Time is divided into rounds; at each round, all clients send their inputs and views of the (latest) contract state to the contract. The contract processes the inputs and moves to the next round only if its local state matches the views received by all clients. A similar approach is used in Lightweight Collective Memory (LCM) [14] where a set of mutually trusted clients interacts with a TEE application and exchange their view of the system to detect inconsistencies.

Vulnerability to Forking Attacks: A fixed set of clients may help prevent forking attacks based on rollbacks. For example, a FastKitten smart contract can detect a rollback if the clients' state information does not match the local state. Similarly, in LCM, clients can detect if any response from the smart contract does not match the client's global view. We note, however, that a solution with a fixed set of clients does not prevent cloning attacks: the adversary may still have an advantage in running multiple clones of the enclave. For example, when the computation is randomized, the adversary can launch multiple clones and forward the more favorable output to the clients.

Limitation 3—Fault tolerance. This approach requires clients to be online and to trust each other. Hence, the application does not tolerate client crashes or byzantine faults.

Limitation 4—Reconfiguration. Reconfiguration operations, i.e., allowing clients to join or leave the set of participating clients, are costly operations that require re-negotiations of all established cryptographic keys in the system.

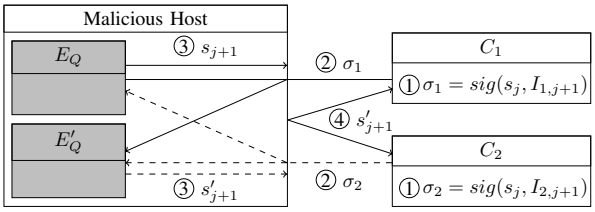

Fig. 4. Example of a cloning attack against a non-deterministic smart contract in FastKitten [8]. A malicious host starts two enclaves and selects the preferred output.

Example: FastKitten. Das *et al.* [8] propose a TEE-based solution to deploy smart contracts on Bitcoin. FastKitten consists of an enclave E_Q that executes a smart contract with a fixed set of clients. During setup, all clients deposit coins to E_Q's address, which are then redistributed during the contract execution. The execution of the smart contract is split into computation rounds. The enclave redistributes the coins in each round j, advancing from state s_{j-1} to s_j. The enclave regularly seals the state to provide fault tolerance. At the beginning of a round, each client i signs the previous state s_{j-1} and its input $I_{i,j}$ for the current round. If the enclave state is rolled back, the local state will not match the state sent by clients. Note, however, that [8] does not address Limitation 4.

Despite being secure against rollback attacks, a cloning attack here is still possible if the enclave executes a probabilistic smart contract (e.g., a lottery contract as suggested in [8]). Assume a malicious host that runs an enclave E_Q that has already completed the setup phase with two clients, as shown in Figure 4. Let the enclave state be s_j and assume it is sealed to disk. The malicious host now starts a clone E'_Q and provides it with s_j. In the next round, each client i binds s_j to its input $I_{i,j+1}$ by means of a signature (step ①). The clients submit their signatures to the host that feeds it to both E_Q and E'_Q (step ②). Both enclaves successfully verify the signatures and treat s_j as the current valid state. At this stage, each enclave computes a different output since the lottery contract is randomized (step ③). The adversary selects the most favorable output (e.g., the output that favors a specific client) and forwards it to the clients (step ④).

> **Takeaway 3 – Fixed Set of Clients.** Relying on a fixed and mutually trusted set of smart contract clients can prevent rollback attacks; however, it cannot prevent cloning attacks if the enclave is non-deterministic.

D. Serializing State

Overview. Most platforms use the blockchain's ordering layer to persist and serialize the enclave state. Across the platforms we analyzed, we witnessed three variants of this technique.

Option 1—Transaction replay from the ledger. Some platforms do not use the local sealing functionality for the enclave to recover the state after a crash. Instead, the enclave recovers state by fetching all blocks from the blockchain and processing one block after another [9], [13], [43], [47], [48]. This option can prevent rollback attacks if the enclave obtains the complete set of blocks up to the current one. It can also protect against cloning if the blockchain cannot be forked.

Option 2—Timestamping. Other platforms use sealing to persist state locally (e.g., on disk) but include block metadata—such as its height and hash—in their state as an anchor to ensure state freshness [1], [9], [32], [43], [45]–[48], [52]–[54]. For instance, the block height—i.e., the number of blocks from the genesis block until the current block—can be used as a logical clock to track which transactions were committed to the ledger and which were processed by the enclave. Here, it is paramount that an enclave includes its current timestamp (e.g., the height of the last block it has processed) in its responses to contract queries. This burdens the requesting client to compare the timestamp in the ledger (i.e., the current block height) with the one included in the response from the enclave to detect forking attacks. Some platforms, such as TERNOA [45], allow clients to specify a range of block heights for their contract queries (to cater to partial-synchronous deployments). Here, a query includes a minimum block height m and a maximum block height M. The enclave serves the query only if the height of the latest processed block falls within $[m, M]$. This design choice requires clients to keep track of the current block height and only ensures loose synchronization between the ledger and the TEE. In particular, an adversary could roll back the enclave to a state where the latest block has not been processed yet to ensure that the TEE's answer does not take a recent transaction into account. In other platforms, such as IntegriTEE [10], the enclave regularly sends a heartbeat transaction to the blockchain, including the current block height. The enclave only answers contract queries if it receives the corresponding acknowledgment.

Option 3—Storing state in the blockchain. Another variant that we witnessed involves enclaves that seal their state (representation) on the ledger [10]–[12], [33]–[35], [37], [38], [51], [54]. For example, the enclave can write the hashes of the input and output states to the blockchain. Hence, the consistent layer can check that the new state naturally evolves from the latest stored state [12]. This strategy can protect against rollback attacks if the enclave can always access the latest state in the consistent layer. It can also prevent cloning attacks since state updates are tied to the previous state and the hash of the enclave code.

In what follows, we discuss several limitations that affect the three options mentioned above.

Limitation 5—Low throughput. Most permissionless L1 layers exhibit low throughput to ensure safety; for instance, Ethereum has an average block interval of 12 seconds[1], while Bitcoin has an average block interval of 10 minutes[2], severely

[1]https://ycharts.com/indicators/ethereum_average_block_time, accessed 27.06.2024

[2]https://studio.glassnode.com/metrics?a=BTC&category=&m=blockchain.BlockIntervalMean, accessed 27.06.2024

limiting the number of state updates that can be performed in a certain time interval (when the state is stored in the blockchain) or the granularity of state updates (when the blockchain is used to timestamp responses).

Limitation 6—Existential honesty. The only means for enclaves to receive all blockchain history (and keep track of the actual block height or hash) is to (1) either directly participate in consensus [47] or (2) connect to at least one *honest* blockchain node (this is often referred to as the existential honesty assumption [55]). Running the entire consensus within an enclave [47] is usually discouraged as it increases the TCB size and code complexity, thereby increasing the attack surface [56]. Without direct participation in the consensus protocols, enclaves need to connect to multiple blockchain nodes (e.g., lightweight Bitcoin clients are recommended to connect to at least four nodes) to ensure that at least one of their connections is honest and faithfully reports the current block. However, this limitation can be mitigated if (1) enclaves tie their response to their local timestamp (e.g., block height, hash), and (2) clients check whether their timestamp matches the one in the enclave's response. In this sense, one can effectively outsource the existential honesty limitation from the enclaves to the clients, who now have to be connected to at least one honest blockchain node to track the current state.

Limitation 7—Blockchain forks. Permissionless systems like Ethereum and Bitcoin only offer eventual consistency; forks (i.e., blocks with the same height) can naturally occur, which, in turn, would weaken the security provisions of this approach to protect TEE states from forking. For example, BITE [1] uses enclaves that scan Bitcoin blocks and answer client requests. Since Bitcoin only provides eventual consensus, assume that a fork occurs in Bitcoin at height h and a light-client LC submitted a transaction t that is only included in one of the forks. Next, LC queries the enclave to determine its balance. A malicious operator can create two clones of the BITE enclave, providing each instance with one of the blockchain forks. LC will receive different balances depending on which clone serves the request. Similarly, Narrator [57] is a TEE-based anti-forking solution for TEE applications; here, a set of Narrator enclaves provide an anti-forking mechanism for enclaves running arbitrary applications. The security of Narrator holds as long as (1) the set of Narrator enclaves is not forked, and (2) each platform runs at most one Narrator enclave. Towards this end, at start time, a Narrator enclave writes a platform-bound ID to a blockchain; this makes it possible to distinguish if two Narrator enclaves are running on the same machine. Clearly, a fork on the blockchain would allow a malicious operator to run two Narrator enclaves on the same machine. An effective countermeasure to address this problem would be to couple enclave responses with the block height *and* the block hash. This allows clients to determine that this state was computed from a fork.

Limitation 8—Randomized computations. There exists another attack avenue on enclaves that execute randomized contracts. In particular, the adversary can run the enclave multiple times—by re-running a single instance repeatedly or by running multiple instances at once—to obtain different outputs, each dependent on the randomness drawn by the enclave instance during the computation. Obtaining different outputs could provide an unfair advantage for the adversary. For example, consider the case where the smart contract outputs a winning lottery ticket. In that case, the adversary can obtain multiple tickets—each output by one of the enclave instances—and decide which one to broadcast as the winning one. Examples of randomized smart contracts that are vulnerable to such attacks include PoUW [33] (cf. Section IV-A), Proof of Luck [34], lottery contracts [8] (cf. Section IV-C), and Ten [12] (cf. Section V-C). An effective solution is to use ephemeral IDs. In particular, if each enclave creates an ephemeral key pair sk, pk at startup and uses sk to sign its output, the adversary cannot obtain multiple outputs that verify with respect to pk. This solution works as long as ephemeral enclave IDs (i.e., their public keys) are appropriately managed. For example, in the lottery application described above, clients must agree on the enclave instance (and its public key) that is entitled to draw the winning ticket.

A note on timestamping and monotonic counters. The idea of using the current block height to tell if the TEE state is fresh is reminiscent of "monotonic counters". Monotonic counters have been proposed (and used) in the context of TEEs to prevent rollback attacks. That is, an enclave can use a monotonic counter to prevent its local state from being rolled back—e.g., once the enclave has processed a transaction tx, the adversary cannot roll back the enclave to a previous state where tx had not been processed. For instance, Milutinovic *et al.* [34] propose monotonic counters to prevent cloning attacks on TEE-based leader elections. Here, an enclave sleeps for a random period and generates a signed *Proof of Luck (PoL)* afterward, which the miner includes in a block proposal. The PoL protocol increments the monotonic counters on a platform before drawing a random number. The enclave validates that the monotonic counters have the expected value before generating the PoL in the last step of the protocol. If an adversary runs multiple PoL enclaves, each enclave will increase the monotonic counter at the beginning of the protocol, and all but one enclave will see a counter mismatch at the end of the protocol. However, a monotonic counter does not help the enclave distinguish whether it has processed all transactions committed to the blockchain. Furthermore, monotonic counters are usually implemented via TPM registers. As such, they represent a single point of failure and tend to wear out [2]. In practice, we note that monotonic counters are not viable solutions against rollback attacks since most platforms no longer support them [58], [59].

Example: CCF. Howard *et al.* [29], [47] propose a framework for permissioned confidential blockchains (CCF). CCF runs PBFT or RAFT within the enclaves as the underlying

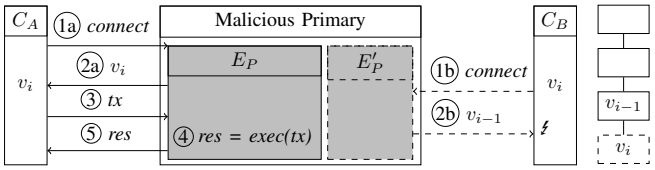

Fig. 5. Example of a (failed) cloning attack on CCF [47]. A malicious primary runs two enclaves, E_P and E'_P, where E'_P keeps an outdated state. C_B detects a view mismatch and terminates the connection.

consensus protocols (permissioned consensus protocols). CCF services are based on a replicated key-value store (KVS). To ensure consistency of the KVS, CCF divides time into a series of views v. In each view, one of the nodes is elected as primary while the other nodes serve as backups. Only the primary enclave E_P processes client transactions tx and appends them to the ledger. The current state of the ledger, i.e., all committed transactions, is represented by a Merkle tree. E_P appends every executed transaction to the Merkle tree, regularly signs the Merkle root of the ledger, and appends it to the ledger as a *signature transaction*. A transaction tx is only considered committed if a signature transaction, including tx, is replicated on most nodes. When the enclave restarts, it recovers the global state by replaying all transactions from the ledger. When a client connects to CCF, it receives the current view v_i, which denotes the current progress in the ledger. Here, v_i serves as a logical timestamp associated with the enclave state, which the client can check before submitting a transaction to the service.

Figure 5 shows how CCF leverages the v_i to prevent cloning attacks. Assume a malicious primary who runs two enclaves E_P and E'_P. It provisions E_P and E'_P with all transactions up to the views v_i and v_{i-1}, respectively. When a client connects to the enclave (step ①), it first returns its current view (step ②). The client checks the provided view against the view it knows. Client C_B, who is connected to E'_P, will detect a mismatch in the views and terminate the connection. Only client C_A, connected to E_P, sends the transaction tx as the views match (step ③). The enclave executes the transaction, computing the result (step ④), and finally returns it to the client (step ⑤). Note that a rollback attack would result in an outdated view, so the client would not provide its transaction to the enclave.

Note that CCF overcomes Limitations 5-8 as follows. First, CCF's consensus algorithm is based on PBFT and RAFT, which provide high throughput and overcome Limitation 5. These consensus protocols also guarantee finality, alleviating Limitation 7. CCF executes the whole consensus algorithm in an enclave. Enclaves have a direct view of the block height (i.e., *view*) and do not need to rely on the honesty of other nodes, thus bypassing Limitation 6. We note, however, that implementing consensus inside the enclave significantly increases the TCB and is not a recommended design choice [56]. Finally, since clients directly connect to the CCF enclave through an encrypted TLS session, a malicious primary cannot inspect different outcomes of a randomized contract execution, alleviating Limitation 8.

Takeaway 4 – Serializing State. Serializing the enclave output using a consistent layer (e.g., the consensus layer of blockchains) can prevent rollback and cloning attacks. However, it needs to be combined with ephemeral IDs to prevent cloning attacks when the TEE computations are non-deterministic.

E. Summary of Findings

Table I presents the results of our study, while Figure 14 in Appendix A offers an additional overview of the distribution of countermeasures. We summarize our findings below.

- Out of the 29 TEE-based blockchain platforms we analyzed, five are vulnerable to forking attacks. In particular, four platforms are vulnerable to cloning attacks, while one is vulnerable to rollback and cloning attacks. In Section V, we focus on three of those five platforms—those that are either fully deployed or offer at least a testnet—and describe the attacks in detail. We stress that none of the attacks we present were known before.
- 16 of the platforms that we considered use stateless enclaves (cf. Section IV-A). 11 of those platforms rely on the untrusted host to provide state information to the enclave [9], [13], [33]–[35], [38], [43], [46]–[48], [50].
- Seven of the analyzed platforms leverage Ephemeral IDs (cf. Section IV-B). Five of these platforms are stateless and secure against forking attacks. Among those platforms, only three platforms [35], [42], [51] require proper key management (cf. Limitation 2). The remaining four platforms do not require key management as the application logic is inherently robust to cloning [39], [41], [43], [45]. For example, [39] runs a mixer in the enclave, and the adversary has no advantage in running multiple clones.
- Two platforms rely on a fixed set of clients or nodes (cf. Section IV-C). As both applications are subject to Limitations 3 and 4, none supports reconfigurations—hence, the set of participating parties is determined at setup time and remains fixed throughout the contract lifetime [8], [40].
- 21 platforms serialize state with one of the three techniques we discussed in Section IV-D.
  - Five applications replay transactions at enclave restart to recover the state (cf. Option 1). Four out of these applications additionally timestamp enclave responses to queries [9], [29], [43], [48]. The Secret Network [13] also replays transactions but does not rely on additional serialization mechanisms, allowing forking attacks.
  - 11 enclaves timestamp their responses to queries (cf. Option 2). BITE [1] and Tesseract [43] use the latest block hash whereas the other nine applications [9], [29], [45], [46], [48], [52]–[54] timestamp by means of the block height.

TABLE I

SUMMARY OF OUR ANALYSIS OF 24 TEE-BASED BLOCKCHAIN APPLICATIONS FROM [16], [31]. WE REPORT FOR EACH APPLICATION WHICH FORKING MITIGATION(S) ARE USED AND WHETHER THEY OVERCOME THE CORRESPONDING LIMITATIONS. WE DENOTE THAT A COUNTERMEASURE IS USED WITH ✓. FURTHER, WE WRITE ✗ (RESP. ✓) IF THE APPLICATION OVERCOMES A LIMITATION AND LEAVE THE FIELD EMPTY IF A LIMITATION IS NOT APPLICABLE BECAUSE THE APPLICATION DOES NOT DEPLOY THE CORRESPONDING COUNTERMEASURE. ⋆ MEANS THAT THE UNDERLYING BLOCKCHAIN IS NOT SPECIFIED IN ENOUGH DETAIL TO REASON ABOUT LIMITATIONS 5-8, RESPECTIVELY.

Project	Forking Mitigations						Limitations							
	Stateless enclaves	Ephemeral identities	Fixed set of clients	Transaction replay	Time-stamping	State on the ledger	Functionality			Robustness				Performance
							L1	L4	L8	L2	L3	L6	L7	L5
TEE-based Smart Contracts														
Azure CCF [47]	✓			✓	✓		✗		✗			✗	✗	✗
CONFIDE [32]					✓				✗			✗	✗	✗
CreDB [52]					✓				⋆			⋆	⋆	⋆
Ekiden [11]						✓			✗			✗	✗	✗
Phala [9]	✓			✓	✓		✗		✗			✗	✓	✗
Secret Network [13]	✓			✓			✗		✗			✓	✗	✗
TEE-based Consensus Protocols														
Crust sWorker [53]					✓				✗			✗	✗	✗
ENGRAFT [35]	✓	✓				✓	✗		✗	✓		✗	✗	✗
MobileCoin [49]	✓						✗							
Proof of Luck [34]	✓					✓	✗		✓			✗	✗	✗
REM [33]	✓					✓	✗		✓			✗	✗	✗
TEE-based Layer 2 Solutions														
COMMITEE [42]	✓	✓					✗			✓				
FastKitten [8]			✓					✓			✓			
Hybridchain [51]		✓				✓			✗	✓		✗	✗	✗
IntegriTEE [60]						✓			✓			✗	✓	✗
Obscuro Mixer [39]	✓	✓					✗			✗				
PrivacyGuard [50]	✓						✗							
Private Chaincode [37]						✓			✓			✗	✗	✗
Private Data Objects [38]	✓					✓	✗		✗			✗	✗	✗
ShadowEth [54]					✓	✓			✗			✗	✗	✗
Teechain [40]			✓					✓			✓			
Ten [12]						✓			✓			✗	✗	✗
Tesseract [43]	✓	✓		✓	✓		✗		✗	✗		✗	✗	✗
Twilight [41]	✓	✓					✗			✗				
TEE-based Blockchain Applications														
BITE [1]					✓				✗			✗	✗	✗
LSKV [48]	✓			✓	✓		✗		✗			✗	✗	✗
sgxwallet [44]	✓						✗							
Ternoa Network [45]		✓			✓		✗		✗	✗		✗	✗	✗
Town Crier [46]	✓				✓		✗		✗			✗	✗	✗

– Finally, ten applications store state on the blockchain (cf Option 3). Three platforms leverage the enclave to generate new blocks [12], [33], [34]. The consensus layer only accepts the block if the included hash is the latest block hash. Similarly, ENGRAFT [35] requires replicas to approve that a state update evolves from the latest state. Six applications directly store their state on the blockchain; the consensus layer ensures linearizability of the state (i.e., that a state can only evolve from the previous state [10], [11], [37], [38], [51], [54]). Only ShadowEth [54] deploys Options 2 and 3 in parallel.
– All 21 applications leverage a high-throughput L1 or batch state updates for on-chain storage (cf. Limitation 5). For example, Tesseract [43] updates client balances in memory and only sends settlement transactions to the L1 once per day. Note that CreDB [52] does not provide enough details about the underlying blockchain to determine if it is subject to any limitation.
– Only one application requires existential honesty (cf. Limitation 6): the Secret Network [13] does not validate the order of transactions the host provides. In CCF [29], the enclaves also actively participate in consensus.
– 18 platforms can tolerate blockchain forks (cf. Limitation 7). Nine applications rely on a fully consistent consensus layer [13], [32], [35], [37], [38], [47], [48], [51], [54] to serialize events. Five additional applications cryptographically bind the output (including the block hash) on chain [11], [12], [33], [34], [45], and two applications make the current block available to their clients [1], [43]. Additional two applications are not subject to blockchain forks due of their logic [46], [53].
– A total of 15 platforms do not allow the exploitation of randomized computations (cf. Limitation 8). Eight of these platforms run deterministic contracts [1], [11], [32], [43], [45], [46], [48], [53]. Another seven platforms encrypt the enclave response for the client so a malicious host cannot read it [9], [13], [29], [35], [38], [51], [54].

• An analysis of the adoption of the mitigation strategies across different system classes is provided in Appendix A.

Summary: We conclude that mitigations for forking attacks introduce trade-offs in terms of (1) types of applications that can be deployed, (2) tolerance to peers joining/leaving the network, and (3) overall complexity of the platform. For example, stateless enclaves can prevent rollback attacks but

limit the type of contracts that can be deployed. For contracts that require a local state, rollback attacks can be mitigated if the state is serialized on L1. The latter can be achieved in several ways. However, the choice of the L1 layer is very important as it determines the effectiveness of the serialization technique. For instance, a low-throughput L1 will result in a coarse-grained timestamping mechanism. We note that ten platforms that serialize enclave state using the L1 layer use permissioned blockchains to achieve high throughput and finality.

With respect to cloning attacks, deterministic smart contracts are safe as long as the platform guarantees that all smart contracts process the same events and in the same order (e.g., by serializing state on L1). If the smart contract is randomized, ephemeral IDs allow to distinguish clones of the same contract but may require proper key management and means to determine, at any time, the set of enclaves that are part of the platform (and their IDs).

V. Case Studies

In this section, we discuss in detail three new cloning attacks on three prominent production-ready TEE-based blockchains: Ten, Phala, and the Secret Network. These case studies serve to exemplify the pitfalls and mitigations discussed in Section IV.

Case Study 1 $\xrightarrow{\text{uses}}$Serialization (IV-D), $\xrightarrow{\text{has}}$ Limitations 6,7,8

Case Study 2 $\xrightarrow{\text{uses}}$Stateless Enc. (IV-A), $\xrightarrow{\text{has}}$Limitations 2,5-7

Case Study 3 $\xrightarrow{\text{uses}}$Serialization (IV-D), $\xrightarrow{\text{has}}$ Limitation 2

We note that all of these blockchains share some design principles. Namely, they all rely on a global secret—shared among all enclaves—to derive cryptographic material, such as encryption keys. For example, enclaves of the Secret Network share a so-called "consensus seed" used to derive, e.g., the public key used by a user to send encrypted requests to the enclave and the corresponding secret key used by enclaves to decrypt those requests. Another common design principle is the reliance on contract queries to save gas and improve latency. Phala and the Secret Network allow users to send contract queries to the enclaves via an HTTP endpoint. Such queries are encrypted using a public key derived from the network-wide secret. Our analysis considers rollback attacks on the sealed global secret out of scope.

We responsibly disclosed our findings and recommendations to the Secret Network, Ten, and Phala, respectively, and shared our suggested countermeasures with their teams.

A. Case Study 1: Phala

Overview: *Phala* [9] is a Layer 1 (L1) blockchain that is built using the Substrate framework [61]. It acts as a para-chain that plugs into *Polkadot* [62]. At the time of writing, Phala has an active mainnet with a market cap of $90M [63], 157 deployed contracts, and 150 Worker nodes serving over 800k off-chain queries per day [64]. Phala leverages TEEs to enable off-chain confidential smart contract execution. Figure 6 shows the components of the Phala Network and how they interact. There are two types of nodes in the Phala Network: *Gatekeepers* and *Workers*. Both nodes operate a *pruntime* enclave connected to the network through a *pherry relayer*. Gatekeepers additionally run a *collator*, which participates in the consensus. Phala uses Authority-round (Aura) consensus, a Proof-of-Authority (PoA) leader election algorithm that elects the next leader to create the next block. At the time of writing, the developers control all leaders in the system. Workers are responsible for smart contract execution. They are assigned specific smart contracts—typically, a contract is run by a single Worker, but multiple Workers for the same contract can be used. Phala derives an asymmetric *contractKey* for each smart contract using a global *masterKey* accessible only to Gatekeeper enclaves. The Worker enclave queries the *contractKey* from the Gatekeeper, seals it, and checkpoints its contract state to disk to provide fault tolerance. When a Worker receives a new block, it parses out all transactions for its contract, decrypts them, and processes them. Further, each Worker has an HTTP endpoint, which clients can use to issue contract queries. Phala incentivizes these contract queries, as they do not cost gas fees and result in fast responses.

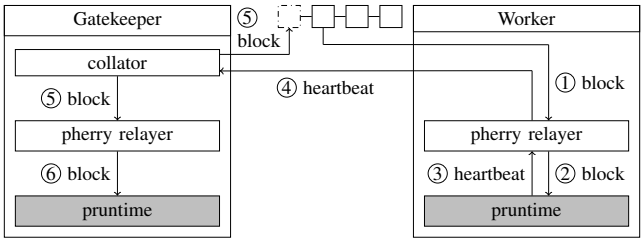

Fig. 6. Overview of the components in Phala [9]. Worker pruntimes regularly issue heartbeat transactions to the ledger, which are checked by Gatekeeper pruntimes.

As each smart contract is only executed by one enclave, Phala implements heartbeats as a keep-alive mechanism, shown in Figure 6. In particular, the Worker's pherry relayer regularly fetches new blocks from the blockchain (step ①) and forwards them to the pruntime (step ②). The pruntime computes a function of blockchain meta-data and its public key hash to determine whether or not to send a heartbeat. The function is designed so that approximately 20 Workers send a heartbeat in each block. In other words, a Worker pruntime issues a heartbeat roughly every 45 seconds. A heartbeat is sent to the pherry relayer as a transaction containing the current block height (step ③). The pherry relayer submits the heartbeat transaction to a Gatekeeper's collator (miner) (step ④) who verifies the transaction and includes it in the next block (step ⑤). The Gatekeeper's pherry relayer fetches the block and forwards it to the Gatekeeper pruntime (step ⑥), which extracts, validates, and logs the heartbeat.

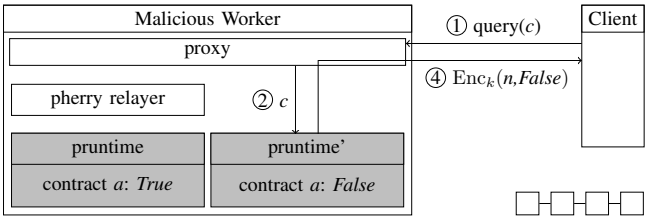

Fig. 7. Sketch of the cloning attack on Phala [9]. A malicious Worker clones the pruntime running the smart contract. It then prevents the clone from receiving state updates and answers contract queries with an outdated state.

```
#[pink::contract(env=PinkEnvironment)]
mod phat_important_data {
    use super::pink;
    use pink::{PinkEnvironment};

    #[ink(storage)] //persistent storage
    pub struct ImportantData {
        data: bool,
    }

    impl ImportantData {
        #[ink(constructor)]
        pub fn new() -> Self {
            Self { data: true }
        }

        #[ink(message)] //on-chain command
        pub fn toggle(&mut self) {
            self.data = !self.data;
        }

        #[ink(message)] //contract query
        pub fn get_data(&self) -> bool {
            self.data
}}}
```

Fig. 8. Example of a smart contract in Phala [9] that persists storage. Clients can change the stored boolean value via on-chain transactions or read the contract state through a contract query.

Cloning Attack on Contract Queries: As shown in Figure 6, Phala enclaves directly receive blocks from the pherry relayers (who fetch them from the ledger). Note that the pherry relayers do not run within the TEEs and, hence, could easily perform rollback attacks on the enclaves by providing a stale state from the ledger. Thus, Phala enclaves issue heartbeat transactions every 45 seconds (cf. Figure 9 for the exact format of the heartbeat messages). While heartbeats contain the block height (and, as such, could be used for timestamping), enclaves do not check whether they receive regular heartbeats (or acknowledgments) from others. This check is performed only by the Gatekeeper (cf. Figure 6) and does not reside within the Worker enclave. This allows enclaves to be cloned and even isolated from the rest of the network.

We note that contract queries are encrypted with a symmetric key k derived from a key exchange that uses an ephemeral key of the client and the *contractPubKey* of the smart contract. In particular, the client computes k as follows:

$$\begin{aligned} k &= \mathrm{ECDHKE}(\mathrm{clientPrivKey}, \mathrm{contractPubKey})) \\ &= \mathrm{ECDHKE}(\mathrm{clientPubKey}, \mathrm{contractPrivKey})) \end{aligned}$$

where ECDHKE is an Elliptic-curve Diffie–Hellman key exchange and clientPrivKey is the ephemeral private key of the client. The client then computes an encrypted query:

$$\begin{aligned} \mathrm{payload} &= \mathrm{AEAD\_IV}||\mathrm{clientPubKey}|| \\ &\quad \mathrm{AES}_k^{GCM}(\mathrm{contractAddress}||\mathrm{n}||\mathrm{rawQuery}) \\ \mathrm{query} &= \mathrm{payload}||\mathrm{clientIdentityPubKey}|| \\ &\quad \mathrm{sign}_{\mathrm{clientIdentityPrivKey}}(\mathrm{payload}) \end{aligned}$$

where AEAD_IV is an IV for the AES encryption, contractAddress is the address of the queried smart contract, clientIdentityPubKey is a persistent client identity for access control, n is a random nonce reflected in the response, and rawQuery is the actual query.

Given the above setting, an adversary can operate two instances of the pruntime and freely choose the instance to answer a request. The attack is depicted in Figure 7. We assume a simple contract with address *a* using a single boolean variable as state, initialized to *False*. At a certain point, a transaction *tx* causes the boolean variable to be set to *True*. From this moment, clients issuing contract queries to the contract at address *a* should receive *True* as a response. However, assume that the adversary creates a clone of the pruntime. The adversary does not start a pherry relayer for the clone, effectively isolating it from the network. The first instance is still connected to the network and issues regular heartbeats. As the cloned pruntime did not receive *tx* (it is isolated from the network), its internal state remains *False*. At this stage, a client issues a contract query (step ①) to the smart contract at address *a*. The adversary forwards the request to the isolated pruntime instance (step ②), which decrypts the query and provides *False* as a response to the client (step ③).

Implementation: We implemented and evaluated the attack on a local Phala Testnet version *v2.1.0* [65] using the official Phala docker images. We stress that no real contract was affected while we were validating our attack and that it had no impact whatsoever on the real Phala Network. The adversary operates a machine equipped with an Intel Xeon E-2286G CPU, 128GB of memory, and Ubuntu 22.04.4 LTS. We configure a single node with a Gatekeeper and a Worker pruntime, each connected to a pherry relayer providing new blocks from the blockchain to the enclave. As for the victim contract, we used a simplified version of the official Phala demo *flip* contract [66], which holds a boolean variable that can be toggled (cf. Figure 8). After initializing the contract to *False*, we start a second Worker pruntime (providing it with the sealed data from the first Worker) and a corresponding pherry relayer. We then terminate the pherry relayer of the second instance, effectively isolating it from the network. We instruct our client to call the deployed smart contract *a*, toggling its state to *True*. The transaction is only processed by the enclave with a connected pherry relayer, such that the isolated enclave remains in state *False*. We then instruct our client to query *a*. At this stage, our proxy intercepts the request and forwards it to the isolated enclave, which returns *False*. Figure 7 shows that the client cannot distinguish which enclave answered the

```
Heartbeat = {
    session_id,
    challenge_block,
    challenge_time,
    iterations,
    n_clusters,
    n_contracts}
```

Fig. 9. Structure of a heartbeat in Phala [9]. The block height is in blue.

request as the reply only contains the state itself and the reflected nonce.

Suggested Countermeasure: Heartbeats in Phala offer a good opportunity to integrate a timestamping mechanism that allows enclaves to self-detect forking attacks: (1) they are authenticated by the enclaves, (2) they contain the block height, and (3) they are sent regularly every 45 seconds. We suggest leveraging those heartbeat messages to ensure that all enclaves are aware of the current block height (and hence the current state). To this end, we suggest that enclaves exchange heartbeats via a separate P2P network and check that they regularly receive heartbeat messages from others (i.e., they are not eclipsed). A major challenge with this approach lies in Limitation 6 (existential honesty): enclaves need to ensure they are connected to at least one honest node to get the latest state from the network (and reflect it in their heartbeat messages).

This, alone, however, is not sufficient to deter forking attacks. Here, we suggest that the enclaves include the latest block height as a timestamp in responses to all contract queries (as suggested in the whitepaper [9]). This burdens the requesting client to determine whether the output corresponds to a fresh state and is, therefore, valid. In case Phala opts to support randomized contract execution, we suggest the reliance on ephemeral IDs (cf. Limitation 8) as well.

B. Case Study 2: The Secret Network

Overview: The Secret Network [13] (SN) is an L1 blockchain that is built using the Cosmos SDK [67]. At the time of writing, it has a market cap of $110M [68] and features 1105 smart contracts uploaded from 197 different accounts.

Each SN node uses a TEE to execute smart contracts. All cryptographic material used in SN is derived from a *consensus seed* shared among all SN enclaves. To obtain the consensus seed, an enrolling enclave must prove—via remote attestation—to one of the SN enclaves that it is running the genuine SN enclave binary. The consensus seed is sealed to disk to avoid re-enrollment in case an SN enclave is restarted.

Transactions to invoke a contract are encrypted with a network-wide public key (called *consensusIoPubKey*) and can be decrypted by any SN enclave. Once a transaction is chosen to be included in the next block, it is decrypted and executed inside the enclave; the contract output is encrypted (e.g., with the sender's public key) and included in the block as well. SN allows contract queries to read the state of the smart contract. Enclaves, in particular, have an HTTP endpoint that the client can use to request the contract state. This design was chosen to avoid gas fees and reduce delays for contract queries.

Cloning Attack on Contract Queries: Previous work [15] has shown that the SN did not feature protection against rollback attacks and that an adversary could learn private data of a transaction (e.g., sender, receiver, amount) by rolling-back the enclave and replaying transactions. In what follows, we present a new forking attack on the Secret Network based on cloning.

We note that contract queries are encrypted with a symmetric key k derived from a key exchange that uses an ephemeral key of the client and the *consensusIoPubKey* of the SN enclave. In particular, the client computes k as follows:

$$\begin{aligned} k' &= \mathrm{ECDHKE}(\mathrm{clientPrivKey}, \mathrm{consensusIoPubKey}) \\ &= \mathrm{ECDHKE}(\mathrm{clientPubKey}, \mathrm{consensusIoPrivKey}) \\ k &= \mathrm{HKDF}(n, k') \end{aligned}$$

where ECDHKE is an Elliptic-curve Diffie–Hellman key exchange, HKDF is a key derivation function, clientPrivKey is the ephemeral private key of the client, and n is a nonce chosen by the client. Note that n and $clientPubKey$ (i.e., the client's ephemeral public key) are sent in cleartext in the transaction to enable the SN enclave to derive the same symmetric key k. We point out that other fields of a transaction are not encrypted—e.g.,the contract address ($contractAddress$) is sent as cleartext. Hence, a query has the following format:

$$\begin{aligned} \mathrm{query} = \; & \mathrm{contractAddress}||\mathrm{n}||\mathrm{clientPubKey}|| \\ & \mathrm{AES}_k^{SIV}(\mathrm{codeHash}||\mathrm{rawQuery}) \end{aligned}$$

where codeHash is the hash of the contract that should handle the query, and rawQuery is the actual query.

Given the above format, an adversary can simply change the contractAddress field of a query and use another instance of the same contract (matching the codeHash in the query) on any network node to answer the query. The attack is depicted in Figure 10. We assume a simple contract with address a that uses a single counter variable as state, initialized to x. At a certain point, a transaction tx causes the variable to be incremented to $\mathrm{x}+1$. From this moment, clients issuing contract queries to the contract at address a should receive $\mathrm{x}+1$ as a response. However, assume that the adversary creates a clone of the contract and assigns it to address *a'*. As the contract at address *a'* did not receive tx (it is a different contract instance), its internal state remains x. Note that the contract enclaves at address *a* and at address *a'* share the same codeHash. At this stage, a client issues a contract query (step ①) for the enclave at address *a*. The adversary intercepts the HTTP request (step ②) and changes the contract address in the requested URL to *a'* (step ③). Hence, the contract enclave at address *a'* decrypts the query and provides x as a response to the client (step ④).

Implementation: We implemented and evaluated the attack on a Secret Network Testnet version *v1.13.1* [69]. We stress that no real contract was affected while we were validating

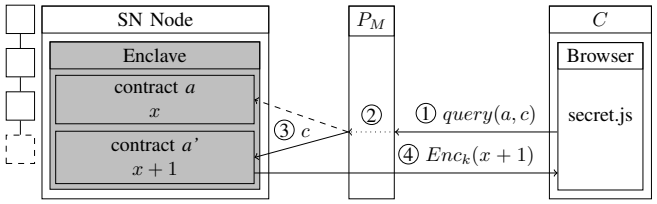

Fig. 10. Sketch of the cloning attack on the Secret Network [13]. A malicious Proxy P_M in the network changes the contract address in the client's query to return the state of a different instance with the same code.

our attack and that it had no impact whatsoever on the real Secret Network. The adversary operates a machine equipped with an Intel Xeon E-2286G CPU, 128GB of memory, and Ubuntu 22.04.4 LTS. The victim uses the Firefox browser version 127.0.2. We configure the Secret Network lightweight client *secretcli* [70] to interact with the testnet. As for the victim contract, we used the official Secret Network contract template [71], which implements a simple counter (cf. Figure 11). We uploaded the contract to the testnet and launched two clone instances of the contract. During the setup, we specified an initial value 1 for the counter on each clone, respectively. We now have two clones of the same smart contract: a contract C_a at address a and another $C_{a'}$ at address a', both with counter value 1. Both contracts are executed by every SN node. We issued a transaction that increments the value of C_a to 2. Note that the transaction includes a (as it targets C_a), so the counter of $C_{a'}$ stays in state 1 although both contracts run in the same enclave. We then built a simple website that uses the official SN lightweight client library *secret.js* [72] to query the state of C_a. Queries are handled via HTTP, so we instantiate a malicious HTTP proxy between our client and the testnet. We let a first query to C_a reach the intended contract so that it returns 2. We then instruct our client to query C_a again. At this stage, the proxy P_M intercepts the HTTP request and replaces address a with a' before feeding it to the enclave, which now returns 1. Figure 10 shows that the client cannot distinguish if the query was answered by C_a or $C_{a'}$ as the reply only contains the state itself.

Suggested Countermeasures: We note that an SN contract is assigned an ID (contract address), but the ID is not bound to the messages exchanged with clients. We note that SN IDs are instance-specific; in other words, two contract clones (same binary, same machine) will get different IDs (similar to the ephemeral IDs we describe in Section IV-B). A straightforward fix to deter cloning attacks would be to cryptographically bind the contract ID to the client request by including it in the encrypted request payload so that the contract instance can tell if it is the intended receiver.

However, the solution just described does not mitigate rollback attacks. Jean-Louis *et al.* [15] suggest implementing a proof-of-publication to ensure that transactions have been committed and ordered on-chain before executing them. This effectively serializes transactions (cf. Takeaway 4). The Secret

```
pub struct InstantiateMsg { pub count: i32,}
pub enum ExecuteMsg { Increment {}, }
pub enum QueryMsg { GetCount {}, }
pub struct CountResponse { pub count: i32, }

#[entry_point] // constructor
pub fn instantiate(deps: DepsMut, _env: Env,
    info: MessageInfo, msg: InstantiateMsg,
)->StdResult<Response> {
    let state = State {
        count: msg.count,
        owner: info.sender.clone(),};
    config(deps.storage).save(&state)?;
    Ok(Response::default())
}

#[entry_point] // on-chain transaction
pub fn execute(deps: DepsMut, env: Env,
    info: MessageInfo, msg: ExecuteMsg
)->StdResult<Response> {
    match msg { ExecuteMsg::Increment {} =>
                        try_increment(deps, env), }
}

pub fn try_increment(deps: DepsMut, _env: Env
)->StdResult<Response> {
    config(deps.storage).update(|mut state| ->
        Result<_,StdError>{state.count+= 1;Ok(state)})?;
    Ok(Response::default())
}

#[entry_point] // contract query
pub fn query(deps: Deps, _env: Env, msg: QueryMsg
)->StdResult<Binary> {
    match msg { QueryMsg::GetCount {} =>
        to_binary(&query_count(deps)?), }
}

fn query_count(deps: Deps)->StdResult<CountResponse> {
    let state = config_read(deps.storage).load()?;
    Ok(CountResponse { count: state.count })
}
```

Fig. 11. Vulnerable smart contract in the Secret Network [71] that persists storage. Clients can increase the stored integer value via on-chain transactions or read the contract state through a contract query.

Network leverages Tendermint, a BFT version of delegated Proof-of-Stake providing some form of finality (cf. Limitation 7) and relatively short block generation times with a throughput of 10000 transactions per second [73] (cf. Limitation 5). However, such an approach would still be limited by the existential honesty assumption (cf. Limitation 6) as the enclave needs to be connected to at least one honest consensus node to ensure access to the latest transactions to be up to date. This is particularly worrisome as Tendermint does not offer strong protection against so-called long-range attacks, where the ledger's history can be rewritten from a past point in time (somewhat analogously to rollback attacks) [22]. A more elegant alternative to deal with rollback attacks against the SN would be to rely on the TEEs to track the set of TEEs (and their ephemeral IDs) that are members of the network (cf. Takeaway 2). This countermeasure, combined with proper reliance on ephemeral IDs as we suggest, would offer a comprehensive solution for the Secret Network against forking attacks.

C. Case Study 3: Ten

Overview: Ten [12] (formerly *Obscuro*) is an L2 solution built on top of Ethereum [74]. At the time of writing, Ten has a public testnet; the mainnet is expected to go live in Q3 2024. Each Ten node uses a TEE to execute transactions in a privacy-preserving manner. By running the Ethereum Virtual Machine (EVM) inside the TEE, any Ethereum smart contract can be ported to Ten. As mentioned earlier, all cryptographic material in Ten is derived from a *master seed* shared among all Ten enclaves. At enrollment, an enclave can obtain the master seed by providing its attestation report. Any Ten enclave can verify this attestation and, when successful, can encrypt the master seed with the enrolling enclave's public key for provisioning. The enclave seals cryptographic key material and smart contract state to avoid re-enrollment when the enclave is restarted. The enclave stores its data, including the *master seed* in *EdgelessDB* [75], a TEE-based SQL database that seals all its data to disk to provide fault tolerance.

Transactions are encrypted with a network-wide public key (called *networkKey*) and can be decrypted by any Ten enclave. Transactions are ordered in the context of *rollups* through a custom *Proof-of-Block-Inclusion (POBI)* consensus protocol. In particular, the enclave extracts the previous rollup from the latest L1 block and generates the next rollup on top of it, including a random nonce. The rollup with the lowest nonce is committed to the L1 layer via a dedicated Ethereum smart contract. Rollup generation uses a throttling mechanism based on proof-of-work to ensure that any Ten enclave can output at most one rollup with its corresponding nonce per block. Ten enclaves are not vulnerable to rollback attacks. This is because rollups are bound to the current L1 block. If the Ten enclave is rolled back, it will output a rollup bound to a stale L1 block; then, L1 will treat the rollup as invalid and discard it.

Cloning Attack on Block Generation: Despite being rollback-resistant, we show that Ten enclaves are vulnerable to cloning attacks that target the POBI consensus protocol. In a nutshell, POBI uses Ten enclaves to draw a random nonce, and the enclave with the lowest nonce is allowed to propose the next rollup. By cloning the enclave, an adversary can increase the chances that one of its enclaves is allowed to propose the next rollup. Ten attempts to combat cloning attacks by requiring a registration fee for parties to enroll their enclaves. In principle, this requires clients to pay the enrollment fee n times when enrolling n enclaves to increase their chances of proposing the next rollup. However, an adversary can circumvent this measure and clone the Ten enclave *after* enrollment. That is, the adversary can create n clones of the Ten enclave while paying the enrollment fee only for one. Note also that this attack is effective, despite the throttling mechanism used by Ten (enclaves must do some variant of proof of work by computing a large number of hashes at restart time) to ensure that each enclave proposes at most one rollup per round. Our attack is depicted in Figure 12 and works as follows. The adversary starts a Ten enclave that completes enrollment, obtains the master seed, and seals

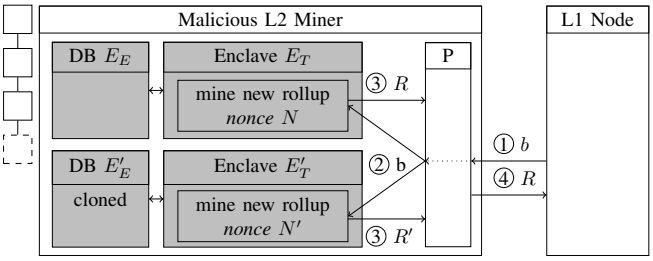

Fig. 12. Sketch of the cloning attack on Ten [12]. An adversary increases the chances of proposing the next block by running two enclave clones and choosing the output with the lowest nonce.

cryptographic keys to disk. Hence, the adversary creates a clone of the enclave on the same machine. At this time, both enclave instances have access to the state sealed by the first enclave. Next, the adversary receives a new block from an L1 node (step ①) and feeds it to both clones (step ②). The enclaves generate random nonces N and N', respectively, and include them in the proposed rollup (step ③). The adversary selects the rollup with the lowest nonce and submits it to the L1 layer (step ④).

Implementation: We implemented and evaluated our attack on a local Ten Testnet version *v0.24.7* [76]. In our setup, the adversary operates a machine equipped with an Intel Pentium Silver J5040 CPU, 32GB of memory, and Ubuntu 22.04.03 LTS. Here, we had to register a Ten node and instantiate an EdgelessDB instance [75] (each running in a docker container) to interact with the testnet. We then cloned the Ten enclave and connected each enclave to a cloned instance of the EdgelessDB to recover the state. We now have two operational Ten enclaves E_T and E'_T on the same node. We then use a simple proxy *P* that handles incoming L1 blocks and feeds them to both Ten enclaves. The enclaves generate the random nonces N and N' and include them in the rollups R and R', respectively. The proxy *P* retrieves R and R' from the enclaves and sends the more favorable rollup back to the L1 layer.[3]

Suggested Countermeasures: Ten implements stateful enclaves (to seal the state of the rollups in an Edgeless DB backend). It incorporates a rollback detection mechanism by serializing state. Here, the state is serialized with the block hash seen by the enclave. If the block hash is stale, the L1 layer will not accept the commit request from the enclave. However, as discussed in Section IV-A, such stateful solutions should also incorporate mechanisms to prevent cloning. An effective anti-cloning mechanism in this particular case would be to rely on ephemeral IDs (cf. Takeaway 2) specific to each enclave. In particular, the L1 contract handling rollups can be easily modified to keep track of the (ephemeral) identities of the TEE enclaves. For instance, the rollup header can include a new field *AggregatorEphemeralID*. Figure 13 shows the rollup header of the current Ten implementation and our

[3]Note that the current implementation of the Ten enclaves is incomplete and does not return the nonces.

```
RollupHeader = {
    L1BlockHeader,
    CrossChainMessages,
    PayloadHash,
    PayloadHashSignature,
    BatchSeqNum,
    AggregatorNonce,
    AggregatorL2Address,
    AggregatorEphemeralID}
```

Fig. 13. Rollup header of Ten [12]. Additional fields based on [77] are shown in blue; our recommendation to incorporate ephemeral IDs is shown in red.

suggested modification to deter cloning attacks. As we discuss in Section IV-B, this would deter cloning attacks.

VI. Conclusion

In this work, we provided a systemization of how current TEE-based blockchains resist forking attacks. To this end, we analyzed 29 TEE-based blockchains and showed an apparent lack of consensus in the community on how to leverage properties from distributed protocols to prevent forking attacks against TEE-based smart contracts. More precisely, we showed that currently used mitigations for forking attacks introduce trade-offs in types of applications that can be deployed, tolerance to peers joining/leaving the network, and overall complexity of the platform.

Our study also revealed new forking vulnerabilities in three production-ready TEE-based blockchains: Phala, the Secret Network, and Ten. We proposed effective countermeasures for each of those vulnerabilities, leveraging the results from our aforementioned analysis. We also responsibly disclosed our findings to the developers of each affected platform.

Acknowledgment

This work is partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and the European Union's Horizon 2020 research and innovation program (REWIRE, Grant Agreement No. 101070627, and ACROSS, Grant Agreement No. 101097122). Views and opinions expressed are, however, those of the authors only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] S. Matetic, K. Wüst, M. Schneider, K. Kostiainen, G. Karame, and S. Capkun, "BITE: Bitcoin lightweight client privacy using trusted execution," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 783–800. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/matetic

[2] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1289–1306. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic

[3] "ZKSwap Finance," [Online; accessed 2. Jul. 2024]. [Online]. Available: https://docs.zkswap.finance

[4] "ZKsync Docs," [Online; accessed 2. Jul. 2024]. [Online]. Available: https://docs.zksync.io/build

[5] "Aztec Network Docs," [Online; accessed 2. Jul. 2024]. [Online]. Available: https://docs.aztec.network

[6] F. Benhamouda, S. Halevi, and T. Halevi, "Supporting private data on hyperledger fabric with secure multiparty computation," in *2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*, A. Chandra, J. Li, Y. Cai, and T. Guo, Eds. IEEE Computer Society, 2018, pp. 357–363. [Online]. Available: https://doi.org/10.1109/IC2E.2018.00069

[7] H. Shrobe, D. L. Shrier, and A. Pentland, *CHAPTER 15 Enigma: Decentralized Computation Platform with Guaranteed Privacy*, 2018, pp. 425–454.

[8] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "FastKitten: Practical smart contracts on bitcoin," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 801–818. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/das

[9] "Phala Network: A Secure Decentralized Cloud Computing Network based on Polkadot," Mar. 2022, [Online; accessed 3. Jul. 2023]. [Online]. Available: https://github.com/Phala-Network/Whitepaper/blob/master/pdf/phala-paper.pdf

[10] "Integritee Network Docs," [Online; accessed 29.06.2024]. [Online]. Available: https://github.com/integritee-network/docs/tree/2f7e468b1a85534ff2da115706f850cf846ae60d

[11] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 185–200.

[12] "https://ten.xyz," Mar. 2024, [Online; accessed 5. Apr. 2024]. [Online]. Available: https://ten.xyz

[13] "Secret Network Graypaper - Secret Network - The Confidential Computing Layer of Web3," Jan. 2024, [Online; accessed 10. Jun. 2024]. [Online]. Available: https://scrt.network/graypaper

[14] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and forking detection for trusted execution environments using lightweight collective memory," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 157–168.

[15] N. Jean-Louis, Y. Li, Y. Ji, H. Malvai, T. Yurek, S. Bellemare, and A. Miller, "SGXonerate:Finding (and Partially Fixing) Privacy Flaws in TEE-based Smart Contract Platforms Without Breaking the TEE," *Proceedings on Privacy Enhancing Technologies*, 2024. [Online]. Available: https://petsymposium.org/popets/2024/popets-2024-0035.php

[16] R. Li, Q. Wang, Q. Wang, D. Galindo, and M. Ryan, "Sok: Tee-assisted confidential smart contract," *Proc. Priv. Enhancing Technol.*, vol. 2022, no. 3, pp. 711–731, 2022. [Online]. Available: https://doi.org/10.56553/popets-2022-0093

[17] G. Doussot, "Smart Contracts Inside SGX Enclaves: Common Security Bug Patterns," Mar. 2020, [Online; accessed 28. Oct. 2024]. [Online]. Available: https://www.fox-it.com/be/research-blog/smart-contracts-inside-sgx-enclaves-common-security-bug-patterns/

[18] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, R. B. Lee and W. Shi, Eds. ACM, 2013, p. 11. [Online]. Available: https://doi.org/10.1145/2487726.2488370

[19] A. Sev-Snp, "Strengthening vm isolation with integrity protection and more," *White Paper, January*, p. 8, 2020.

[20] A. ARM, "Security technology building a secure system using trustzone technology (white paper)," *ARM Limited*, 2009.

[21] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, p. 86, 2016. [Online]. Available: http://eprint.iacr.org/2016/086

[22] E. Deirmentzoglou, G. Papakyriakopoulos, and C. Patsakis, "A survey on long-range attacks for proof of stake protocols," *IEEE Access*, vol. 7, pp. 28 712–28 725, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2901858

[23] S. K. Mohanty and S. Tripathy, "Flexipcn: Flexible payment channel network," in *Financial Cryptography and Data Security. FC 2023 International Workshops - Voting, CoDecFin, DeFi, WTSC, Bol, Brač, Croatia, May 5, 2023, Revised Selected Papers*, ser. Lecture Notes

in Computer Science, A. Essex, S. Matsuo, O. Kulyk, L. Gudgeon, A. Klages-Mundt, D. Perez, S. Werner, A. Bracciali, and G. Goodell, Eds., vol. 13953. Springer, 2023, pp. 405–419. [Online]. Available: https://doi.org/10.1007/978-3-031-48806-1_26

[24] P. Sheng, R. Rana, S. Bala, H. Tyagi, and P. Viswanath, "Proof of diligence: Cryptoeconomic security for rollups," in *6th Conference on Advances in Financial Technologies, AFT 2024, September 23-25, 2024, Vienna, Austria*, ser. LIPIcs, R. Böhme and L. Kiffer, Eds., vol. 316. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 5:1–5:24. [Online]. Available: https://doi.org/10.4230/LIPIcs.AFT.2024.5

[25] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, "zkbridge: Trustless cross-chain bridges made practical," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 3003–3017. [Online]. Available: https://doi.org/10.1145/3548606.3560652

[26] S. A. K. Thyagarajan, G. Malavolta, and P. Moreno-Sanchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 1299–1316. [Online]. Available: https://doi.org/10.1109/SP46214.2022.9833731

[27] A. Gervais, S. Capkun, G. O. Karame, and D. Gruber, "On the privacy provisions of bloom filters in lightweight bitcoin clients," in *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, C. N. P. Jr., A. Hahn, K. R. B. Butler, and M. Sherr, Eds. ACM, 2014, pp. 326–335, dBLP:conf/acsac/GervaisCKG14. [Online]. Available: https://doi.org/10.1145/2664243.2664267

[28] G. Gutoski and D. Stebila, "Hierarchical deterministic bitcoin wallets that tolerate key leakage," in *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, ser. Lecture Notes in Computer Science, R. Böhme and T. Okamoto, Eds., vol. 8975. Springer, 2015, pp. 497–504, dBLP:conf/fc/GutoskiS15. [Online]. Available: https://doi.org/10.1007/978-3-662-47854-7_31

[29] H. Howard, F. Alder, E. Ashton, A. Chamayou, S. Clebsch, M. Costa, A. Delignat-Lavaud, C. Fournet, A. Jeffery, M. Kerner, F. Kounelis, M. A. Kuppe, J. Maffre, M. Russinovich, and C. M. Wintersteiger, "Confidential consortium framework: Secure multiparty applications with confidentiality, integrity, and high availability," 2023. [Online]. Available: https://arxiv.org/abs/2310.11559

[30] C. Müller, M. Brandenburger, C. Cachin, P. Felber, C. Göttel, and V. Schiavoni, "Tz4fabric: Executing smart contracts with ARM trustzone : (practical experience report)," in *International Symposium on Reliable Distributed Systems, SRDS 2020, Shanghai, China, September 21-24, 2020*. IEEE, 2020, pp. 31–40, dBLP:conf/srds/MullerBCFGS20. [Online]. Available: https://doi.org/10.1109/SRDS51746.2020.00011

[31] "Awesome SGX Open Source Projects," 2019. [Online]. Available: https://github.com/Maxul/Awesome-SGX-Open-Source/commit/3f7c58e64e476800900eea2d13b4e7a402ba3c96

[32] Y. Yan, C. Wei, X. Guo, X. Lu, X. Zheng, Q. Liu, C. Zhou, X. Song, B. Zhao, H. Zhang, and G. Jiang, "Confidentiality support over financial grade consortium blockchain," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 2227–2240, dBLP:conf/sigmod/YanWGLZLZSZZJ20. [Online]. Available: https://doi.org/10.1145/3318464.3386127

[33] F. Zhang, I. Eyal, R. Escriva, A. Juels, and R. V. Renesse, "REM: Resource-Efficient mining for blockchains," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1427–1444. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/zhang

[34] M. Milutinovic, W. He, H. Wu, and M. Kanwal, "Proof of luck: An efficient blockchain consensus protocol," in *proceedings of the 1st Workshop on System Software for Trusted Execution*, 2016, pp. 1–6.

[35] W. Wang, S. Deng, J. Niu, M. K. Reiter, and Y. Zhang, "Engraft: Enclave-guarded raft on byzantine faulty nodes," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2841–2855. [Online]. Available: https://doi.org/10.1145/3548606.3560639

[36] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, "Efficient byzantine fault-tolerance," *IEEE Trans. Computers*, vol. 62, no. 1, pp. 16–30, 2013, dBLP:journals/tc/VeroneseCBLV13. [Online]. Available: https://doi.org/10.1109/TC.2011.221

[37] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, "Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric," 2018. [Online]. Available: https://arxiv.org/abs/1805.08541

[38] M. Bowman, A. Miele, M. Steiner, and B. Vavala, "Private data objects: an overview," 2018. [Online]. Available: https://arxiv.org/abs/1807.05686

[39] M. Tran, L. Luu, M. S. Kang, I. Bentov, and P. Saxena, "Obscuro: A bitcoin mixer using trusted execution environments," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 692–701. [Online]. Available: https://doi.org/10.1145/3274694.3274750

[40] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. R. Pietzuch, "Teechain: a secure payment network with asynchronous blockchain access," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds. ACM, 2019, pp. 63–79. [Online]. Available: https://doi.org/10.1145/3341301.3359627

[41] M. Dotan, S. Tochner, A. Zohar, and Y. Gilad, "Twilight: A differentially private payment channel network," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 555–570. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/dotan

[42] A. Erwig, S. Faust, S. Riahi, and T. Stöckert, "Commitee : An efficient and secure commit-chain protocol using tees," in *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*. IEEE, 2023, pp. 429–448, dBLP:conf/eurosp/ErwigFRS23. [Online]. Available: https://doi.org/10.1109/EuroSP57164.2023.00033

[43] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels, "Tesseract: Real-time cryptocurrency exchange using trusted hardware," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 1521–1538. [Online]. Available: https://doi.org/10.1145/3319535.3363221

[44] "sgxwallet: SKALE SGX-based hardware crypto wallet." [Online]. Available: https://github.com/skalenetwork/sgxwallet

[45] "Ternoa Documentation," [Online; accessed 18. Jun. 2024]. [Online]. Available: https://docs.ternoa.network

[46] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 270–282. [Online]. Available: https://doi.org/10.1145/2976749.2978326

[47] M. Russinovich, E. Ashton, C. Avanessians, M. Castro, A. Chamayou, S. Clebsch, M. Costa, C. Fournet, M. Kerner, S. Krishna *et al.*, "Ccf: A framework for building confidential verifiable replicated services," *Microsoft, Redmond, WA, USA, Tech. Rep. MSR-TR-2019-16*, 2019.

[48] "Lskv: Democratising confidential computing from the core," Feb. 2023, [Online; accessed 07. Jun. 2024]. [Online]. Available: https://archive.fosdem.org/2023/schedule/event/cc_lskv/

[49] MobileCoin, "MobileCoin Whitepaper," Aug. 2023, [Online; accessed 6. Jul. 2023]. [Online]. Available: https://assets-global.website-files.com/654151be5c8f07ebc15119fb/65de65639faa4449d5a9d5c8_MobileCoin-Whitepaper-2023-08-03.pdf

[50] Y. Xiao, N. Zhang, J. Li, W. Lou, and Y. T. Hou, "Privacyguard: Enforcing private data usage control with blockchain and attested off-chain contract execution," in *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II*, ser. Lecture Notes in Computer Science, L. Chen, N. Li, K. Liang, and S. A. Schneider, Eds., vol. 12309. Springer, 2020, pp. 610–629, dBLP:conf/esorics/XiaoZLLH20. [Online]. Available: https://doi.org/10.1007/978-3-030-59013-0_30

[51] Y. Wang, J. Li, S. Zhao, and F. Yu, "Hybridchain: A novel architecture for confidentiality-preserving and performant permissioned blockchain using trusted execution environment," *IEEE Access*, vol. 8, pp.

190 652–190 662, 2020, dBLP:journals/access/WangLZY20a. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.3031889
[52] K. Mast, L. Chen, and E. G. Sirer, "A vision for autonomous blockchains backed by secure hardware," in *Proceedings of the 4th Workshop on System Software for Trusted Execution*, ser. SysTEX '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3342559.3365333
[53] "Crust White Paper v1.9.9," Nov. 2020. [Online]. Available: https://ipfs.io/ipfs/QmP9WqDYhreSuv5KJWzWVKZXJ4hc7y9fUdwC4u23SmqL6t
[54] R. Yuan, Y. Xia, H. Chen, B. Zang, and J. Xie, "Shadoweth: Private smart contract on public blockchain," *J. Comput. Sci. Technol.*, vol. 33, no. 3, pp. 542–556, 2018, dBLP:journals/jcst/YuanXCZX18. [Online]. Available: https://doi.org/10.1007/s11390-018-1839-y
[55] J. A. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," *J. ACM*, vol. 71, no. 4, pp. 25:1–25:49, 2024. [Online]. Available: https://doi.org/10.1145/3653445
[56] I. X. P. Intel, "Intel software guard extensions developer guide," 2017. [Online]. Available: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://cdrdv2-public.intel.com/671581/intel-sgx-developer-guide.pdf&ved=2ahUKEwjT67jHp5qHAxXAnf0HHThbBb8QFnoECBkQAQ&usg=AOvVaw2mQhBDlvoKa6chdZ1u0y9P
[57] J. Niu, W. Peng, X. Zhang, and Y. Zhang, "NARRATOR: secure and practical state continuity for trusted execution in the cloud," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 2385–2399. [Online]. Available: https://doi.org/10.1145/3548606.3560620
[58] "SGX PSE Intel Community post," Mar. 2019, [Online; accessed 9. Jul. 2024]. [Online]. Available: https://community.intel.com/t5/Intel-Software-Guard-Extensions/Platform-Service-Enclave-and-ME-for-Intel-Xeon-Server/m-p/1173100/highlight/true
[59] "Sgx monotonic counter alternatives," Jul. 2023, [Online; accessed 9. Jul. 2024]. [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000057968/software/intel-security-products.html
[60] "Integritee Litepaper," [Online; accessed 29.06.2024]. [Online]. Available: https://www.integritee.network/docs/Integritee_%20Lightpaper_2021.pdf
[61] "https://substrate.io," Nov. 2023, [Online; accessed 13. Nov. 2023]. [Online]. Available: https://substrate.io
[62] "https://polkadot.network," Nov. 2023, [Online; accessed 13. Nov. 2023]. [Online]. Available: https://polkadot.network
[63] "Phala Network," Jun. 2024, [Online; accessed 29. Jun. 2024]. [Online]. Available: https://coinmarketcap.com/currencies/phala-network
[64] "Phala 2023: Year in Review," Jul. 2024, [Online; accessed 8. Jul. 2024]. [Online]. Available: https://phala.network/posts/phala-2023-year-in-review
[65] "Release PRuntime v2.1.0 · Phala-Network/phala-blockchain," Jul. 2024, [Online; accessed 8. Jul. 2024]. [Online]. Available: https://github.com/Phala-Network/phala-blockchain/releases/tag/pruntime-v2.1.0
[66] "phat-contract-examples/flip/lib.rs at master · Phala-Network/phat-contract-examples," Jul. 2024, [Online; accessed 2. Jul. 2024]. [Online]. Available: https://github.com/Phala-Network/phat-contract-examples/tree/master/flip
[67] "Cosmos SDK Docs," Jul. 2024, [Online; accessed 3. Jul. 2024]. [Online]. Available: https://docs.cosmos.network
[68] "Secret," Jun. 2024, [Online; accessed 17. Jun. 2024]. [Online]. Available: https://coinmarketcap.com/currencies/secret
[69] "Release v1.13.1 · scrtlabs/SecretNetwork," Jul. 2024, [Online; accessed 8. Jul. 2024]. [Online]. Available: https://github.com/scrtlabs/SecretNetwork/releases/tag/v1.13.1
[70] "Releases · scrtlabs/SecretNetwork," Jul. 2024, [Online; accessed 1. Jul. 2024]. [Online]. Available: https://github.com/scrtlabs/SecretNetwork/releases/tag/v1.13.1
[71] "secret-template," Jul. 2024, [Online; accessed 1. Jul. 2024]. [Online]. Available: https://github.com/scrtlabs/secret-template
[72] "secret.js," Jul. 2024, [Online; accessed 1. Jul. 2024]. [Online]. Available: https://github.com/scrtlabs/secret.js
[73] "Secret Network Benchmarks," [Online; accessed 7. Jul. 2024]. [Online]. Available: https://scrt.network/about/about-secret-network/#:~:text=Delegated%20Proof%2Dof%2DStake%20consensus,of%20only%206%2D7%20seconds.
[74] "Ethereum Whitepaper," Mar. 2024, [Online; accessed 30. Jun. 2024]. [Online]. Available: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf
[75] "https://www.edgeless.systems," Apr. 2024, [Online; accessed 6. Apr. 2024]. [Online]. Available: https://www.edgeless.systems
[76] "Release Obscuro v0.24.7 · ten-protocol/go-ten," Jul. 2024, [Online; accessed 8. Jul. 2024]. [Online]. Available: https://github.com/ten-protocol/go-ten/releases/tag/v0.24.7
[77] "Obscuro Whitepaper," Nov. 2021, [Online; accessed 3. Jul. 2023]. [Online]. Available: https://whitepaper.obscu.ro

APPENDIX

A. Distribution of Mitigation Strategies

We now analyze the adoption of the mitigation strategies discussed in Section IV across the different system categories identified in Section III-B. Figure 14 depicts the distribution of these mitigations across the different system categories. The following key observations can be made based on our results:

- **Stateless enclaves** are employed by platforms in all four categories.
- **Ephemeral identities** are primarily used by platforms in Category 3 (TEE-based Layer 2 solutions) to counter cloning attacks. Specifically, five out of 13 platforms in Category 3 utilize ephemeral IDs. In contrast, no platform in Category 1 (TEE-based smart contracts) uses ephemeral identities.
- **A fixed set of clients** is a technique exclusively employed by platforms in Category 3.
- **State serialization** techniques are utilized across all four system categories. Platforms in Category 1 mostly rely on transaction replay and timestamping, whereas Category 2 and 3 more commonly store their states on the ledger. Many platforms in Category 4 (TEE-based blockchain applications) use timestamping. Lastly, no platform in Category 2 replays transactions to recover state information, and no enclave in a blockchain application stores its state on the ledger.

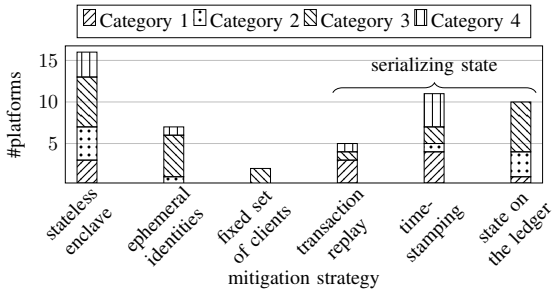

Fig. 14. Distribution of the mitigation strategies used by the platforms in Table I. Note that platforms may use more than one strategy to prevent forking.