Multi-Certificate Attacks Against Proof-of-Elapsed-Time And Their Countermeasures

Huibo Wang
Baidu Security
wanghuibo01@baidu.com

Guoxing Chen
Shanghai Jiao Tong University
guoxingchen@sjtu.edu.cn

Yinqian Zhang[†]✉
SUSTech
yinqianz@acm.org

Zhiqiang Lin
Ohio State University
zlin@cse.ohio-state.edu

***Abstract*—Proof-of-Elapsed-Time (PoET) is a blockchain consensus protocol in which each participating node is required to wait for the passage of a specified time duration before it can participate in the block leader election in each round. It relies on trusted execution environments, such as Intel SGX, to ensure its security, and has been implemented in Hyperledger Sawtooth and used in many real-world settings. This paper examines the security issues including fairness guarantees of the Sawtooth's PoET design and implementation, and discovers a new category of security attacks against PoET, dubbed Multi-Certificate Attacks, which allows a malicious node to unfairly create multiple Certificates in each round of block leader election and select the one that maximizes her probability of winning. We have systematically analyzed the root causes of these attacks and assisted the Sawtooth community to fix several vulnerabilities in the latest version of PoET. To further mitigate the identified threats, we propose a new design of PoET in this paper, which we call PoETA, that can be used to address the remaining vulnerabilities we have discovered. We have implemented PoETA and evaluated its security and performance.**

I. Introduction

The most important technology in blockchain, a decentralized and append-only ledger, is the consensus protocol, which maintains the consistency of the views of the ledger from all nodes in the network, even in the presence of dishonest participants. Over the past decade, numerous consensus protocols have been proposed, from the earliest Proof-of-Work (PoW), which underpins Bitcoin [35], to a variety of alternative designs, such as Proof-of-Elapsed-Time (PoET) [5], Proof-of-Ownership [22], Proof-of-Burn [36], Proof-of-Capacity [19], Proof-of-Luck [32], Proof-of-Stake [25], Proof-of-Activity [11], Proof-of-Reputation [21], and so forth.

Among these consensus protocols, PoET is of particular interest due to its scalability and potential for wide adoption. In particular, PoET is a production-ready consensus protocol supported by the Hyperledger Sawtooth project [3]. Compared with Bitcoin and Ethererum, Hyperledger Sawtooth is designed to be adaptable and customizable; it serves diverse use-cases on different scales; it allows on-chain control of network settings and policies so that it can be used for permissioned and closed environments. Hyperledger Sawtooth is applicable to seafood chain traceability, bond asset settlement, and marketplace digital asset exchanges [2]. In practice, Hyperledger Sawtooth has already been used by Blockchain Technology Partners (BTP) [1], a leading enterprise blockchain company that combines Hyperledger Sawtooth with Kubernetes to create an enterprise blockchain management platform for simplifying enterprise blockchain adoption, and in ScanTrust [6] for bringing transparency to supply chains.

Meanwhile, unlike many other consensus protocols, PoET uses a *proof-of-elapsed-time* to select the block leader. A proof-of-elapsed-time is a proof that a participating node samples a random variable and waits until an amount of time specified by the sample has elapsed. The participating node with the smallest sample, *i.e.*, "elapsed time" becomes the leader to mine a block to be appended to the chain. To prevent nodes from cheating, a trusted execution environment (TEE), such as Intel SGX, is introduced to sample the random variable, and generate a proof-of-elapsed-time (in the form of a Certificate) after checking the sampled amount of time has elapsed.

Consequently, the authenticity of the Certificates plays a central role in ensuring two main properties of the protocol:

- **Fairness**: The "elapsed time" is not controlled by the node and is generated in a (pseudo)random manner, such that each node would have the same probability of winning the leader election.
- **Trustworthiness**: The node indeed waits for the "elapsed time" specified in the Certificate for the current round of leader election.

Fundamentally, since Certificates are generated inside TEEs, they are believed to be protected from dishonest nodes, who will benefit from manipulating Certificates to increase their probabilities of winning the block elections.

In this paper, we examine whether TEEs like SGX are indeed capable of providing guarantees of such *fairness* and *trustworthiness* for consensus protocols like PoET. Surprisingly, our study has led to the discovery of a new category of attacks against SGX-based PoET, which we call *Multi-Certificate Attacks (MCA)*. In a Multi-Certificate Attack, the adversary, a malicious node in the blockchain network, could generate multiple Certificates during the same round of block leader election and then select the one that maximizes the

[†]Yinqian Zhang is affiliated with the Department of Computer Science and Engineering and Research Institute of Trustworthy Autonomous Systems of Southern University of Science and Technology (SUSTech).

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24158
www.ndss-symposium.org

likelihood that the adversary will win the election either in the current round or in future rounds. MCAs allow the adversary who controls one or multiple nodes of the network to breach the fairness of PoET and mine more blocks than her fair share. MCAs may enable easy-to-perform selfish-mining attacks against PoET blockchains.

Moreover, we have found two variants of MCAs: parallel MCAs and sequential MCAs. In parallel MCAs, multiple Certificates are generated in parallel, without the node waiting for the designated elapsed time. We have identified vulnerabilities in Hyperledger Sawtooth v1.0.5, which enable two types of parallel MCAs. Specifically, the first vulnerability allows the adversary to generate multiple values of the "elapsed time" and choose the shortest one to wait and then create the corresponding Certificate. The second vulnerability allows the adversary to generate multiple Certificates for the same value of the "elapsed time" and pick the one with the maximum likelihood to win in the next round. We have demonstrated successful attacks against Sawtooth v1.0.5 and reported the vulnerabilities to the developers of Sawtooth, and received positive responses with acknowledgments. These vulnerabilities have also been patched in Sawtooth v1.1 [7].

In sequential MCAs, multiple certificates are generated one after another, with the randomly generated wait time indeed elapsing before generating each certificate. Although only a limited number of Certificates can be generated in each round, sequential MCAs still offer the adversary vantage points to subvert the fairness of the protocol. The root cause of such attacks is that *the expected elapsed time is revealed to the adversary once it is created, rather than after the required time elapses*. Hence, the adversary knows whether the value of the timer is in her favor, and adjusts her strategy to launch attacks accordingly. Sequential MCAs are still effective on the patched version of Sawtooth v1.1, as it is non-trivial to eliminate such vulnerability in the design of PoET.

To mitigate sequential MCAs, we propose a new design of PoET, and we name it PoETA where "A" stands for "Afterward", as the fundamental idea of our design is to reveal the expected elapsed time of the wait timer "after" the required time elapses. Particularly, we introduce a *determine-after-check* approach to reveal a portion of the expected elapsed time (dubbed a duration segment), and determine whether the expected elapsed time is reached after each duration segment elapses. As such, no participant (honest or not) could predict the value of the expected elapsed time in advance. Meanwhile, PoETA has the identical distribution of the elapsed time as PoET (and thus the block generation rates), which is designed with several properties required for proper distribution of consensus. We have implemented PoETA and evaluated it with a number of benchmarks. Our evaluations show that PoETA provides optimal performance overhead with a stronger security guarantee.

Contributions. In short, we make two main contributions:

- **New Attacks against PoET (§V).** We perform a systematic study of security issues of PoET and demonstrate a new category of attacks, Multi-Certificate Attacks, that compromise fairness properties of Sawtooth PoET. Two vulnerabilities causing MCAs have been reported and patched in Sawtooth v1.1.
- **A Secure Design of PoET (§VI).** We propose a new secure design PoETA for Sawtooth, with concrete implementation and evaluation. Our experimental results show that PoETA has optimal performance overhead with stronger security and meanwhile the same expected elapsed time distribution as PoET.

Roadmap. The rest of this paper is organized as follows. In §II, we provide necessary background related to Intel SGX and PoET consensus. Then §III provides an overview of PoET and security analysis of the fairness property of the PoET protocol. §IV presents our threat model. Next, we discuss real-world vulnerabilities we have identified in the PoET design and implementation of Intel's Sawtooth 1.0, and analyze SGX-based PoET in general in §V. We then investigate the problems in Sawtooth PoET design and propose the new design PoETA in §VI, and its implementation and evaluation in §VII. In §VIII, we discuss a straightforward method of eliminating replay attacks, an alternative design of PoETA, potential attacks to other TEE settings, and cases with multiple malicious nodes, followed by reviewing the related work in §IX. Finally, §X concludes the paper.

II. Background

A. Intel Software Guard Extension

Intel SGX aims at protecting sensitive data from privileged software such as operating system and hypervisor, by providing a shielded execution environment, called *enclave*, to protect the confidentiality and integrity of code and data loaded inside [31], [17]. Adopting SGX typically requires applications to be partitioned into trusted and untrusted components. The trusted components contain the sensitive application data (e.g., secrets) and operations that work on those data. To facilitate SGX application development, Intel SGX SDK provides an enclave definition language (EDL) to define an enclave's trusted and untrusted interface functions, along with tools to process EDL to create proxy or bridge code to call into (ECall) and return from (OCall) an enclave [23].

Remote Attestation. Intel SGX provides a mechanism, named *remote attestation*, to enable a remote party to establish trust in an enclave. It provides a proof that a specific enclave has been correctly instantiated and is running on a real SGX platform, and that the remote party is securely communicating with this enclave. The remote party can verify whether the running enclave's identity matches its expectations. Intel adopts Enhanced Privacy ID (EPID), a group signature scheme based on an anonymous signature technique, to preserve SGX platform privacy [24]. EPID supports two modes of attestation service: linkable quotes and unlinkable quotes. Linkable quotes allow one to distinguish whether two given signatures are from the same or different signers, whereas unlinkable quotes do not.

Sealing. Intel SGX also provides a mechanism, called *sealing*, for enclaves to seal secrets and store them outside of the enclave, *e.g.*, on persistent storage. With sealing, secrets could be restored when the enclave is torn down. Encryption is performed with a private sealing key, which is unique to

the platform and enclave. Sealing keys can be bound to two types of identities: (i) Enclave Identity, when sealing with the Enclave Identity, the sealed data could only be unsealed by the identical enclave; (ii) Signing Identity, sealing with the Signing Identity allows the sealed data to be unsealed by different enclaves as long as they are signed by the same developer.

Monotonic Counters. Intel SGX leverages trusted monotonic counters supported by Intel Management Engine (ME) to prevent rollback attacks, *i.e.*, attacks that replace the current enclave state with outdated states. Intel provides a privileged enclave, called Platform Service Enclave (PSE) to facilitate enclaves to access these counters. Notably, each counter is associated with a counter ID and a counter value. The counter ID is used to identify different counters, and the counter value is used to identify the version of secret data for the verification of the freshness of data. An enclave could register a counter with the PSE, and then read, increase the counter value. The counter accessibility is decided by the owner's policy, such as enclaves with the same signing key, enclaves with the same measurement, or enclaves with both the same signing key and measurement. Since these counters are stored in a non-volatile memory located in Intel ME, updating the value of a counter is time-consuming (around 80-250 ms) [29].

B. Blockchain

A blockchain is a decentralized and self-regulating peer-to-peer system without a central authority. A consensus protocol is needed to achieve agreement on a value or state of a blockchain among multiple participants. The consensus protocol is designed to be fair, reliable, and secure. One main problem in the consensus protocol is called Byzantine Generals problem [26], which has been addressed by the classic Byzantine fault tolerance (BFT) algorithms [14]. However, classic BFT algorithms run into scalability problems and permissioned BFT algorithms only fit for a closed group or semi-closed of known nodes in small scales.

Interestingly, Nakamoto consensus [34] used in the Bitcoin network offers an alternative solution to this problem, which shows that it is possible to form a consensus in a peer-to-peer network open to anonymous nodes. It proposes the idea of the Proof-of-Work (PoW) consensus [35] mechanism, which replaces the leader election in the closed network. Nakamoto consensus includes a set of rules governing the network's authenticity in conjunction with the PoW mining consensus. It can be mainly broken into three parts: PoW consensus (or Nakamoto consensus), block validation, and incentive mechanism. Specifically, PoW involves solving a cryptographic puzzle with unavoidable CPU power consumption. As such, the longest chain represents the majority decision since it consumes the greatest computational resources. As the mining process is stochastic, the PoW consensus works like a fair lottery game with an equal chance of winning for each player. During the block validation, the first node that solves the puzzle will propose a new block to the network and be the winner if all the transactions in the block are valid. The incentive mechanism is that each node competes in the mining block to earn a block reward when the miner successfully mines a validated block. The incentive mechanism helps encourage nodes to stay honest and spend efforts validating and securing the network.

Blockchain Forks. In a blockchain, if two miners mine two different blocks at roughly the same time, the chain will be forked into two branches. Other miners in the network may continue to work on either branch. These two branches are called chain forks. Forking is one of the most critical problems in distributed networks. Forks are inevitable due to the decentralized nature of blockchain networks, *e.g.*, propagation delay, and connectivity quality. For each fork, there is a group of miners working on it. The group with more computing power tends to dominate the blockchain. The proper fork resolver should suggest the miners join the group with more computing power to increase the chance that the blockchain will accept its blocks. In PoW, the fork resolving strategy adopts the longest chain that is formed by costing more mining power or more nodes. In case two fork chains are of the same length, the strategy is to attach to the first fork chain's nodes.

When an attacker or a group of attackers could control more than 50% mining power in the blockchain network, they are in control of the blockchain network. Therefore, they can select any fork, prevent new transactions from gaining confirmations, and/or reverse transactions that were completed, which means that they can do double-spending. This attack is called 51% attack [10].

Selfish Mining. While controlling 51% mining power can be challenging, if not impossible, Eyal *et al.* proposed another type of attack, called selfish mining, which requires relatively less mining power [20]. The critical idea of selfish mining is that the malicious nodes, called selfish miners, intentionally fork the chain by keeping the discovered block private in a pool shared by them. Instead of mining on the public chain, the selfish miners mine on their private chain, while the honest miners are mining on the public chain. When the private chain becomes longer than the public chain, the selfish miners publish their private chain. Being the longest chain, the private chain will be accepted by honest miners and become the public chain eventually. The blocks mined by the honest miner previously will be dumped. This strategy will cause honest miners to waste their resources and lead them to leave the blockchain network. It may also defer the inclusion of legitimate transactions into the blockchain, creating opportunities for blockchain attacks such as double-spending.

C. Hyperledger Sawtooth and PoET

Hyperledger Sawtooth is an enterprise blockchain platform composed of two levels [4]: an application level and a core system level. Application developers can specify the business rules appropriate for their applications without knowing the design of the core system. Also, Hyperledger sawtooth is highly modular, so the users can define policy rules, including transaction rules, permissions, and consensus algorithms that support their business needs. Furthermore, various consensus protocols are supported, including Sawtooth PBFT, Sawtooth Raft, and SGX-based PoET. Particularly, for SGX-based PoET, Hyperledger Sawtooth utilizes the trusted execution environment of Intel SGX to reduce the power consumption of Nakamoto consensuses such as PoW, by performing leader elections using trusted programs running inside SGX enclaves. The integrity of enclave programs can be verified through SGX remote attestation. In addition, PoET is a generic consensus

protocol that can be used by any federated blockchain platform other than Hyperledger Sawtooth.

III. Overview of Sawtooth PoET

In this section, we provide an overview of the Sawtooth PoET. At a high level, PoET works as follows: First, each node in the network generates a timer denoted by waitTimer, and then goes to sleep for a pseudorandom expected elapsed time denoted by duration before waitTimer expires; Then, each node broadcasts its duration to the network for block leader election. The node with the shortest duration may commit a new block to the blockchain. As such, the security of PoET depends on two critical properties: (i) *Fairness*: every node has an equal chance of winning the election. In other words, the duration cannot be manipulated by the node. (ii) *Trustworthiness*: the winner has indeed waited for an amount of time denoted by duration before block leader election.

Sawtooth leverages Intel SGX to achieve these two properties. More specifically, fairness can be achieved by protecting the waitTimer and duration inside an enclave such that the node could not manipulate them. Trustworthiness can be achieved by generating a Certificate inside an enclave that is signed by its private key, as long as the trusted enclave code verifies that a node does wait for the duration time until the expiration of the waitTimer.

A. Sign-up and Election Phases

In the Sawtooth implementation of the consensus protocol, a node has two roles: a client and a server, depending on the execution phases of the consensus protocol. At a high level, there are two phases of the PoET consensus: the sign-up phase and the election phase.

- **Sign-up Phase.** The sign-up phase is the initial phase of the consensus, and it has two steps as well. First, a new participating node (client) downloads and runs the SGX enclave code. A pair of asymmetric keys are generated inside the enclave. The participating node then requests to join the blockchain network by broadcasting its sign-up certificate (including its public key) with an SGX attestation quote. Second, the existing nodes in the network (servers) verify the join request. If the attestation is accepted, the sign-up certificate of the new node will be recorded by each node in the network.
- **Election Phase.** After sign-up, the new node can participate in the block election. The election phase utilizes the concept of proof-of-elapsed-time. Specifically, each node in the network creates a waitTimer inside an SGX enclave. A random duration is sampled from a specific distribution inside the enclave to ensure that the node cannot manipulate it. The node waits outside the enclave until the duration has passed. Then it calls into the enclave, which checks if the waitTimer has expired. If so, a Certificate signed by the node's private key is produced by the enclave and broadcast to the entire blockchain network. The node with the shortest duration in each round is elected as the leader who could publish the block.

B. The Two SGX ECalls for Block Election

Two ECalls are implemented inside the SGX enclaves for the main functionality of the election phase:

- **`ecall_CreateWaitTimer()`.** This ECall creates a trusted timer with its expiration time (*i.e.*, duration) randomly sampled from a specified distribution. Because there is no trusted hardware timer supported directly by SGX, a software timer must be implemented inside an SGX enclave. To do so, `ecall_CreateWaitTimer()` uses the `sgx_get_trusted_time` API (with *second*-level precision) provided by Intel's SGX SDK as a trusted clock. This API is implemented as a platform service offered by the Platform Service Enclave (PSE), which returns the value of the Real-Time Clock protected by Intel Management Engine (ME).

  To guarantee freshness of the Certificate, a node must prove that a trusted timer is created for the current round of block leader election. Therefore, information regarding the current blockhead must be included in the creation of the timer. Additional information regarding the node can also be included as input to this ECall.
- **`ecall_CreateWaitCertificate()`.** This ECall generates a cryptographic proof stating waitTimer was created and has expired. Since the enclave code no longer runs after `ecall_CreateWaitTimer()` returns, the node will not be notified when the timer expires. Hence, the node must call into the enclave again to check if the timer has expired. This functionality can be achieved by the `ecall_CreateWaitCertificate()` ECall, which checks if the trusted timer has expired by invoking the `sgx_get_trusted_time` API again and, if so, continues to create a certificate signed by the enclave's private key.

C. Distribution of duration

At the core of a PoET protocol design is the proof of the elapsed time. In particular, the elapsed time that a node must wait is treated as a random variable, duration. When duration is sampled from an exponential distribution, the block generation can be modeled as a Poisson process. The exponential distribution should have a mean that is large enough, such that in expectation, very few nodes will produce blocks at roughly the same time. If multiple blocks are indeed generated in the same epoch, the dispute can be resolved by selecting the one with a shorter elapsed time. The mean value should not be too large either, such that more blocks can be generated per unit time. Sampling the distribution should provide insights into the size of the population, which will help adjust duration according to the population to avoid forks.

D. Fairness of PoET

Ensuring fairness is one of the most important tasks in a blockchain. In PoET, ensuring fairness requires that each participating node has an equal probability of winning a block leader election. To prevent a single node from gaining a higher winning probability than others, Sawtooth PoET performs Z-test [8] to determine if a node wins the block election with a probability higher than expectation. In other words, Z-test is to

detect compromised SGX platforms by limiting the frequency of winning elections for each node. However, our attack could still break the fairness as long as the frequency of winning elections by malicious nodes is within the limit (which is larger than the expected/average frequency).

Without fairness, malicious nodes may perform *51% attacks* or *selfish mining attacks* on the PoET blockchain, by creating a chain that is more likely to win during fork resolution before releasing it to the honest nodes. In theory, the fork resolver selects a chain with more computing resources involved. In PoET, the fork resolver selects the fork chain with larger aggregate localMean, which is an estimate of the total amount of time spent on waiting. Specifically, localMean of i-th round is calculated as

$$\text{localMean}_i = \text{targetWaitTime} \times \text{populationSize}_{i-1} \quad (1)$$

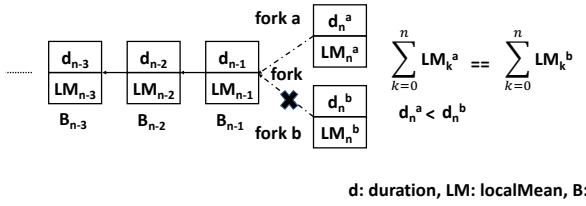
where targetWaitTime is the expected wait time of the wining block and $\text{populationSize}_{i-1}$ is an estimate of the number of nodes in the network based on previous $i-1$ blocks. When two fork chains have the same aggregate localMean, the resolver attaches to the chain the block with a shorter duration.

The strategy of fork resolution in PoET is shown in Figure 1. Specifically, as presented in Equation 1, the localMean of each block is calculated by the product of the targetWaitTime and the estimate of $\text{populationSize}_{i-1}$ based on previous blocks on the chain. Therefore, two forks' chain heads with the same previous blocks will eventually have the same localMean, which directly leads to the same aggregated localMean for the two forks as shown in Figure 1a. In this case, the fork a with the smaller duration is selected. Whereas two forks' chain heads having different previous blocks will result in different localMeans, and thus different aggregated localMean as shown in Figure 1b. In this case, the fork a with the larger aggregated localMean is selected.

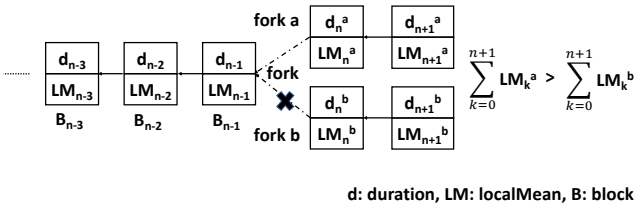
E. Existing Security Mechanisms

To prevent a malicious node from abusing the consensus protocol, Sawtooth adopts the following security mechanisms:

- **Linkable Signature.** To stop each node from signing up multiple enclaves at the same time, Sawtooth protocol enforces the use of linkable signature [4] in the attestation quote, so that existing nodes in the network can recognize sign-up requests from different enclaves running on the same SGX machine. Duplicated sign-up requests will replace the old ones. Moreover, after sign-up, a node can only participate in the election phase after c blocks have been generated. As such, a node that failed in the election cannot immediately compete for block leader election in the same round by creating a new enclave to sign up in the network again.
- **Random Nonce.** A nonce generated from a random source (*i.e.*, `rdrand`) inside the enclave is included in a Certificate. With the introduction of randomness, a node cannot control the entire content of Certificate. This mechanism is used to prevent a malicious node from manipulating the content of the Certificate and hence affecting the next round of block leader election, since the winning Certificate is used to create duration of the next round.



(a) When two forks have the same aggregated localMean, the one with a smaller duration will be selected.



(b) When fork chains have different aggregated localMean, the one with the larger aggregated localMean will be selected.

Fig. 1: Fork resolution in PoET.

- **Secure Monotonic Counter.** To prevent an enclave from creating multiple Certificates, using the same enclave instance or multiple enclave instances, the enclave checks a secure monotonic counter [4] and generates the Certificate only if the counter value is as expected. It increments the counter by one each time a Certificate is generated. Moreover, it binds the enclave to a monotonic counter with a specific ID (determined in the sign-up phase) to ensure all enclave instances on the same platform using the same counter.

IV. Threat Model

We assume that the adversary controls a number of dishonest nodes participating in the PoET-based blockchain network. Our analysis will start with only one dishonest node and will be extended to cases with multiple compromised nodes. All software components of the malicious nodes outside the PoET enclaves are managed and controlled by the adversary. As such, the adversary is capable of controlling the CPU scheduling, interrupt delivery and handling, memory management, I/O operations of the malicious nodes.

Intel SGX is used in the PoET protocol and is assumed to be *secure* in terms of both its integrity and confidentiality. That is, the dishonest nodes cannot modify or inspect the code or data inside the PoET enclaves. In addition, trusted timer and monotonic counter services are trusted, i.e., the adversary could not manipulate the clock ticks and counter values. We assume side-channel attacks, including cache side-channel attacks [33] and controlled-channel attacks [41], against the enclave program's memory access patterns, are out of scope. We assume the speculative execution attacks, such as SgxPectre [15], Foreshadow [37], RIDL [39], CacheOut [40], and LVI [38] are mitigated in hardware. Hardware vendors are

```
1 sgx_key_128bit_t     key = { 0 };
2 sgx_key_request_t    key_request = { 0 };
3 key_request.key_name  = SGX_KEYSELECT_SEAL;
4 key_request.key_policy  =
      SGX_KEYPOLICY_MRENCLAVE;
5 sgx_status_t ret = sgx_get_key(&key_request,
       &key);
6 sgx_cmac_128bit_tag_t tag = { 0 };
7 sgx_rijndael128_cmac_msg(key,
      validatorAddress+previousCertificateId,
      &tag);
8 duration =  MINIMUM_WAIT_TIME - localMean *
      log((&tag) / ULLONG_MAX);
```

Fig. 2: Calculating duration for `ecall_CreateWaitTimer()`.

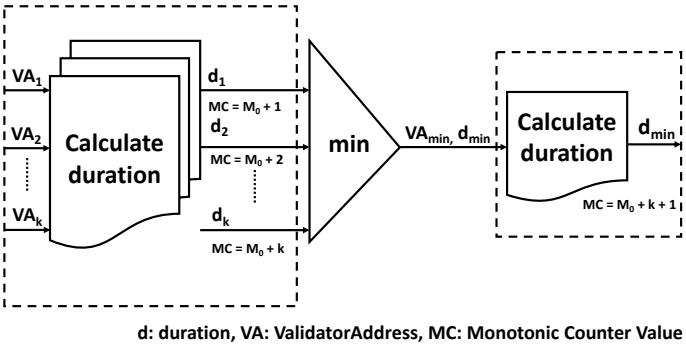


Fig. 3: Workflow of the Parallel MCA against Current Round.

ultimately responsible for eliminating such threats to allow any secure applications to run on the hardware.

V. Multi-Certificate Attacks against Sawtooth PoET

In this section, we describe Multi-Certificate Attacks (MCA) against Sawtooth PoET. In a Multi-Certificate Attack, an adversary generates multiple Certificates during each election round, and picks one that increases the probability of her winning the election in either the current round or the next round.

A. Parallel MCAs

We first present a variant of Multi-Certificate Attacks, which we call Parallel MCAs. In a Parallel MCA, the adversary could produce multiple durations during the same elapsed time. Specifically, we describe two categories of Parallel MCAs. One allows the adversary to select a duration to win the current round of the election with a higher probability by controlling the input to `ecall_CreateWaitTimer()` (§V-A1); the other allows the adversary to create multiple Certificates and select the one to win the next round of the election with a higher probability by abusing the enclave code's use of monotonic counters and secure timers (§V-A2).

1) Parallel MCA against the Current Round: As shown in Figure 2, duration is derived in `ecall_CreateWaitTimer()` from the following variables:

- `validatorAddress`, which specifies the address of the node with the default value being the public key of the node;
- `previousCertificateId`, which is the ID of the Certificate of current block head;
- `localMean`, which is the average of the winning duration of each block in the blockchain and also a field of waitTimer that will be checked by other nodes in the network during block validation.
- `SGX_SEAL`, which is derived from the root secret of SGX (cannot be altered by software);
- `MINIMUM_WAIT_TIME`, which is a constant that specifies the minimum value for duration.

Since the `SGX_SEAL` key and `validatorAddress` vary in each node, the nodes in the network will generate waitTimers with different durations. However, the `validatorAddress` is provided by the node as an argument of the ECall `ecall_CreateWaitTimer()`. Moreover, unlike other arguments of the ECall, its validity is not checked by other nodes of the network. Therefore, it is possible for a malicious node to manipulate the value of `validatorAddress` and affect the calculated duration. Because the node is expected to wait (*e.g.*, sleep) outside the enclave for duration time, the value of duration is returned by the ECall to the node. As such, this implementation is vulnerable as the node is able to choose a `validatorAddress` so that the returned duration is short and the winning probability is high.

Attack Demonstration. The vulnerable code was found in Hyperledger Sawtooth v1.0.5. We have demonstrated the attack on a Sawtooth network with three nodes, each of which ran Ubuntu Linux 16.04 on a 4-core Intel Xeon E-2174G CPU with 16 GB memory and 1Gbps Ethernet Connection. As shown in Figure 3, a malicious node can call `ecall_CreateWaitTimer()` repeatedly with different `validatorAddress` as input until a sufficiently short duration is generated, which is then used to generate a Certificate. It is worthwhile noting that every time `ecall_CreateWaitTimer()` is called, the previously generated durations are no longer valid, as the monotonic counter has been increased by this ECall, which invalidates these duration values. To pass the check of the monotonic counter, the attacker needs to call `ecall_CreateWaitTimer()` again with the corresponding `validatorAddress` to generate the desired duration.

Mitigation. We reported the vulnerabilities to the Sawtooth community in June 2019. The core team acknowledged our findings and released a patch to address the issue. The patch was integrated into Sawtooth v1.1, which ensures that `validatorAddress` used to generate duration is recorded in the Certificate, and signed by the enclave's private key, so that this value can be validated by other nodes in the network. A valid value of `validatorAddress` is expected to be the public key of the enclave that has been registered in the network. As such, the malicious node cannot provide a fake `validatorAddress` to control the value of duration.

```
poet_err_t ecall_CreateWaitCertificate("
    arguments")
{
    uint32_t sequenceId = 0;
    ret = sgx_read_monotonic_counter(
              &validatorSignupData.counterId,
     &sequenceId);
    sp::ThrowSgxError(ret, "Failed to read
    monotonic counter.");

    if (sequenceId != waitTimer.SequenceId)
    {
        Log(
            POET_LOG_ERROR,
            "WaitTimer out of sequence.  %d
    != %d (Attempted replay "
            "attack?)",
            sequenceId,
            waitTimer.SequenceId );
        throw
            sp::ValueError(
                "WaitTimer out of sequence.
     (Attempted replay "
                "attack?)");
    }
    sgx_time_source_nonce_t timeNonce;
    double currentTime = ceil(static_cast<
    double>(GetCurrentTime(&timeNonce)));

    double expireTime =
        floor(waitTimer.SgxRequestTime +
    waitTimer.Duration);

    uint8_t nonce[
    WAIT_CERTIFICATE_NONCE_LENGTH];
    ret = sgx_read_rand(nonce, sizeof(nonce)
    );

    ret = sgx_increment_monotonic_counter(
        &validatorSignupData.counterId,
        &sequenceId);
}
```

Fig. 4: Code Snippet of `ecall_CreateWaitCertificate()`.

2) Parallel MCAs against the Next Round: The enclave code in the `ecall_CreateWaitCertificate()` ECall first checks if a Certificate has been generated before, by checking if the monotonic counter value matches the value passed in as an argument of the ECall. If the check passes, it reads the secure clock (using `GetCurrentTime()`) to determine if waitTimer has expired. If so, a Certificate could be created. Finally, the monotonic counter is increased by 1, indicating a Certificate has been created. The code snippet of the ECall is shown in Figure 4.

However, the code shown in Figure 4 is vulnerable. Specifically, `ecall_CreateWaitCertificate()` has 3 steps: (1) check counter, (2) read secure clock, (3) increment counter. The root cause of this vulnerability is that such an ECall cannot be executed atomically. Step (2) can be paused by a malicious node, as the `GetCurrentTime()`

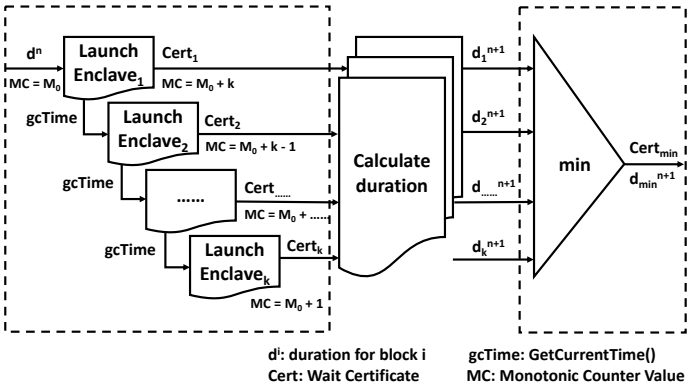


Fig. 5: Workflow of Parallel MCA against Next Round.

API is implemented by the Platform Service Enclave (PSE) and communication between enclaves (the PoET enclave and PSE) can be delayed (though not readable by the adversary as it is encrypted). Therefore, if the malicious node pauses the return of `GetCurrentTime()` API and, in the meantime, creates another enclave instance, she can call into the `ecall_CreateWaitCertificate()` of the second enclave and still pass the check of the monotonic counter, which has not yet been changed by the previous ECall. As such, more than one Certificates can be created.

Attack Demonstration. We have demonstrated an attack against the vulnerable code in Hyperledger Sawtooth v1.0.5 under the same setting described in the previous section. Particularly, at a certain round of blockchain leader election, with the generated duration value, as shown in Figure 5, the node can perform the following steps to create multiple Certificates with the same duration value.

First, `ecall_CreateWaitCertificate()` is called. Then, when the enclave code triggers the `GetCurrentTime()` API, which generates an inter-enclave communication with the PSE, the node suspends the API call before it reaches the PSE. Next, the attacker creates a second enclave with the same enclave binary, and then invokes another ECall `ecall_CreateWaitCertificate()`. Because the first ECall has been paused at the `GetCurrentTime()` call, the monotonic counter has not been increased yet. Consequently, the second ECall to `ecall_CreateWaitCertificate()` passes the check of the monotonic counters. Again, the node suspends the outgoing API call to `GetCurrentTime()`. The adversary can repeat the same procedure by creating more enclaves. Given a particular `duration` value, the malicious node may have the opportunity of creating hundreds or thousands of enclaves. When the wait duration has passed, the node resumes the `GetCurrentTime()` API calls for all these enclaves. Each of these ECalls will finish and return a valid Certificate.

As the Certificate contains a nonce that is different every time a certificate is created, the Certificates created are different and have distinct `WaitCertificateId`s. If the malicious node wins in the current round, the `WaitCertificateId` generated in the current round will be used by all nodes in the blockchain to calculate the duration in the next round. There-

fore, the malicious node has the incentive to choose one of the `WaitCertificateId`s that helps her win in the next round.

Mitigation. We reported the vulnerabilities to the Sawtooth community in June 2019. The core team acknowledged our findings and released a patch to address the issues. The patch was integrated into Sawtooth v1.1, which prevents the attacker from calling `ecall_CreateWaitCertificate()` multiple times for the same elapsed duration. To do so, the value of the monotonic counter is incremented at the beginning of the ECall. In this way, the second call to `ecall_CreateWaitCertificate()` will be rejected immediately as the check for the monotonic counter value fails. By combining these two changes, the Sawtooth patch ensures that only one Certificate can be generated for the same elapsed duration.

B. Sequential MCAs

While both vulnerabilities described above have been fixed in Sawtooth v1.1, this up-to-date implementation still exposes potential vulnerabilities, which enable Sequential MCAs. Specifically, the patch only guarantees that to generate one Certificate, the node must wait outside the enclave for at least duration time. Nonetheless, attackers can still generate more than one Certificates in the same round. This is because to synchronize all nodes in the network entirely, the protocol must accept all Certificates generated within a long-enough time window to tolerate network latency. This time window is what we call the *attack window*. In practice, as the attack window is long enough to encompass multiple small durations, a malicious node may have enough time to generate multiple Certificates (though all of them have the same duration for competing the block leader in the current round) and call `ecall_CreateWaitTimer()` with each of them to *preview* the durations for the next round, assuming the current duration is small enough to be selected as the winner by the network. Then the malicious node can choose the Certificate that increases her likelihood of winning in the next round (should she win the current round) and submit it to the network.

1) Analysis of Attack Windows: Let W be the attack window. To analyze the security of PoET with regard to W, we compute the probability of creating multiple Certificates within W.

Consider an example blockchain network with T nodes. Without loss of generality, we assume the T-th node is malicious, and the rest are honest. In the Sawtooth blockchain design, when a block i needs to be published, all nodes try to publish the block i. Due to the network latency that the certificate with the shortest time maybe come in late, the blockchain will keep updating the block i with the incoming certificate from different nodes which has a shorter time until the block $i+1$ is decided. Let X_t^i denote the duration of t-th node in i-th round. The attack window length for the T-th (malicious) node should be the shortest current-round wait time plus the shortest next-round wait time among the other T-1 nodes because only the current-round block will be in the chain during this time window, the next round would not be in the chain yet. Therefore, we can get the length of the attack window as

$$W = \big(min(X_1^i, ..., X_{T-1}^i) + min(X_1^{i+1}, ..., X_{T-1}^{i+1})\big) \quad (2)$$

TABLE I: Original and manipulated winning probabilities.

Validator Numbers (T)	Original Winning Probability (%)	Manipulated Winning Probability (%)
100	1	29
1,000	0.1	3
10,000	0.01	0.3
100,000	0.001	0.03

Note that each honest node has a winning probability of $p = \frac{1}{T}$. Even if one honest node wins the current round, its winning probability in the next round is still $\frac{1}{T}$. In contrast, when the malicious node wins the current round and try to replay its current round $K-1$ times during the attack window W to generate in total K different Certificates, she could obtain K different durations for the next round, increasing her winning probability to $p' = \frac{K}{T+K-1}$ for the next round. We then analyze the statistical properties of K and then study the adversary's winning probability of p'.

Let

$$A = min(X_1^n, \ldots, X_{T-1}^n)$$

and

$$B = min(X_1^{n+1}, \ldots, X_{T-1}^{n+1}),$$

the probability that the adversary could generate K Certificates given that she wins the current round is

$$P[K] = Pr\Big(KX_T^n < A + B | X_T^n < A\Big) \quad (3)$$

So the expectation of the winning probability in the next round becomes

$$E(p') = \sum_{K=1}^{\infty} \frac{K}{T+K-1} P[K] \quad (4)$$

We represented $P[K]$ in the form of integrals. Details can be found in Appendix A. We use numerical integration to estimating probability. Particularly, we compute the probabilities with four different number of nodes, *i.e.*, $T = 100, 1000, 10,000, 100,000$. The other parameters are set as follows: `MINIMUM_WAIT_TIME` = 1.0, `POPULATION_ESTIMATE_SAMPLE_SIZE` = 50, `TARGET_WAIT_TIME` = 20.0, `BlockSize` = 80,000, `INITIAL_WAIT_TIME` = 5.0. The results are shown in Figure 6. From the result, we can see the probabilities of generating multiple Certificates are not affected by the node size. The probability of generating 2 Certificates is about 73 percent, generating 3 Certificates is about 53 percent, generating 4 Certificates is about 40 percent, the distribution of dropping is as exponential.

We also estimate the expectation of p' accordingly, and the results are shown in Table I. Again, this vulnerability is enabled by the fact that duration is revealed to the node as soon as `ecall_CreateWaitTimer()` returns, which leaves the node an opportunity to check its value and predict whether the value will be in her favor. Consequently, we propose PoETA in the next section to address the problem.

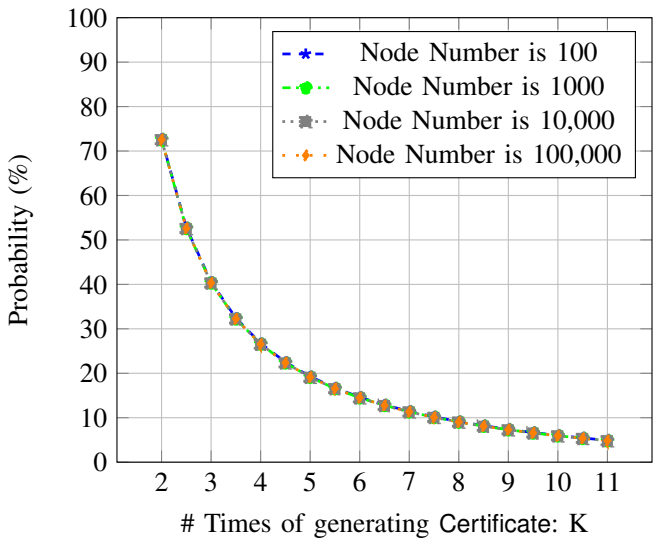


Fig. 6: The probabilities of (K-1) times replay attack with validator numbers as 100, 1000, 10,000, and 100,000

VI. PoETA: A Secure PoET Design

In this section, we present PoETA, a secure design and implementation of PoET, which is proposed to address the problem that causes the Sequential MCAs.

A. Design Goals

As analyzed in §V-B, one key enabling factor of Sequential MCAs is that the value of duration can be previewed so that a malicious node could peek multiple durations of the next round without actually spending time waiting, thus increasing its winning probability for the next round. Hence, we aim to eliminate this key enabling factor to improve the existing PoET design. The design goals of PoETA are as follows.

- **G1-***Security:* The adversary's capability of previewing the duration should be revoked. That is, the final duration should not be revealed until the required time elapses.
- **G2-***Consistency:* The distribution of the duration should be consistent with that of Sawtooth PoET. PoET adopted a distribution of duration with consideration of a stable block generation rate with various population sizes. The new design should follow the same rules to obtain a similar distribution of duration.
- **G3-***Performance:* The overhead of the new design should be kept minimal. Since one of the major advantages of using PoET is to save energy, the new design should preserve this property.

B. Design of PoETA

Now we start to describe the design of PoETA. Particularly, we explain our design choices step by step to address each of the above three design goals.

1) **Concealing duration via *determine-after-check*:** In the original PoET design, duration is finalized and revealed before the required time starts to elapse. This enables the adversary to skip `ecall_CreateWaitCertificate()` if the duration is not in her favor, without any penalty—waiting for the required amount of time. To address **G1**, PoETA must conceal duration until it expires. However, there are two difficulties.

First, it is inevitable to disclose partial information about the duration to the host program, so that it knows when to check back whether the elapsed time reaches the duration, as constantly checking the expiration status might be quite resource-consuming, leading to the degradation of PoET to something like PoW. Second, it is desired to defer the calculation of duration to the latest possible time, eliminating the possibility of leaking the value of duration.

We propose a *determine-after-check* approach, which splits the duration into multiple segments and discloses one segment at a time to the host program. The final duration is generated by accumulating these segments in the following way:

1. When calling `ecall_CreateWaitTimer()`, the first duration segment τ_1, which is sampled from a probability distribution whose probability density function is $f_\tau(x)$, as well as the `MINIMUM_WAIT_TIME` μ, are revealed to the host program, indicating the total amount of time the host program needs to wait before calling `ecall_CreateWaitCertificate()`.
2. During the i-th ($i = 1, 2, \ldots$) invocation of `ecall_CreateWaitCertificate()` after the required time elapses, an oracle $\Theta(i)$ is used to determine whether the final duration is reached (when $\Theta(i) = 1$) or not (when $\Theta(i) = 0$). In the latter case, another duration segment τ_{i+1} will be sampled and revealed to the host program. This step repeats until the oracle returns 1.
3. When the oracle finally returns 1 on the k-th invocation of `ecall_CreateWaitCertificate()`, the duration is finalized and the Certificate is generated.

The final duration $\mathcal{D}$ of PoETA is

$$\mathcal{D} = \mu + \Delta, \tag{5}$$

where $\Delta = \sum_{i=1}^{k} \tau_i$, $\Theta(i) = 0, i = 1, 2, \ldots, k-1$ and $\Theta(k) = 1, k \in \mathbb{N}^+$.

The choices of the duration segment distribution $f_\tau(x)$ and the oracle function $\Theta(i)$ will be detailed later when addressing the second design goal as they are highly related to the distribution of the duration. The state machine of PoETA is shown in Figure 7.

Security Analysis. We show that the duration is revealed after the required time elapses. Note that the client could get a Certificate only when the oracle function call in the current invocation of `ecall_CreateWaitCertificate()` outputs 1. Otherwise, if $\Theta(i) = 0$, the client has to wait for another duration segment, and the duration keeps accumulating. As the output of $\Theta(i)$ is unpredictable, the duration is also unpredictable before it elapses.

While predicting the exact duration is infeasible, the adversary might try to determine whether the final duration is larger than certain threshold Γ so that it is unlikely the smallest duration for the current block. When the adversary learns that the final duration is larger than the threshold, she could start exploring alternatives. In the original design, the adversary knows

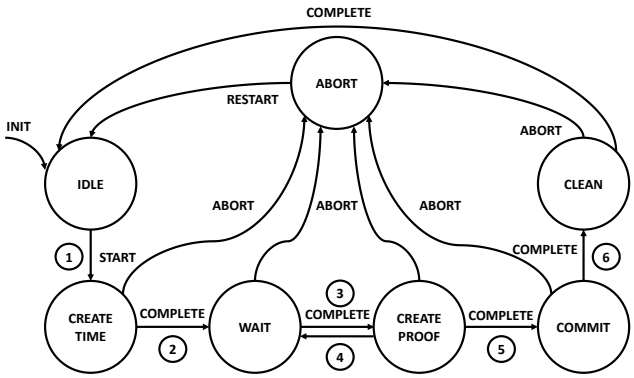


Fig. 7: State Machine of PoETA. ①:New batches of transaction coming, ②:Generated duration, ③: Time elapses, ④: Oracle function $\Theta()$ outputs 0, need to wait one more segment, ⑤: Oracle function $\Theta()$ outputs 1, no need to wait one more segment, ⑥: Broadcast the Proof to the blockchain network.

it immediately after calling `ecall_CreateWaitTimer()`. In PoETA, the adversary might need to wait for several duration segments before she could learn it. We will provide a more detailed analysis in §VI-B3, as it is about the trade-off between security and performance.

2) **Achieving the Identical Distribution of the duration as PoET:** The intuition to address the second design goal is to properly choose the duration segment distribution $f_\tau(x)$ and the oracle function $\Theta(i)$ such that the distribution of the duration is as close to that of the original (which already satisfies those characteristics as discussed in §III-C) as possible. Interestingly, we find the final distribution of duration can be identical to that of the original, through both theoretical analysis and simulations.

As shown in Figure 2, the duration of the original design follows an exponential distribution, with a probability density function

$$f_{\text{PoET}}(x) = \frac{1}{\Lambda} e^{-\frac{1}{\Lambda}(x-\mu)} \tag{6}$$

where Λ and μ denote the localMean and `MINIMUM_WAIT_TIME`. One key property of an exponential distribution is *memorylessness*, *i.e.*, the probabilities at any time of one process are not affected by the history of the process:

$$Pr(\mathcal{D} > s + t | \mathcal{D} > s) = Pr(\mathcal{D} > t), \forall s, t \geq 0$$

When it comes to PoETA, the memorylessness property suggests that each time the oracle function $\Theta(i)$ is called, the probabilities of outputting 1 and 0 should be constant. Hence, we choose the oracle function in PoETA to be a Bernoulli distribution such that

$$\begin{aligned} Pr[\Theta(i) = 1] &= \theta \\ Pr[\Theta(i) = 0] &= 1 - \theta \end{aligned}$$

where $\theta \in [0, 1]$.

Now we need to choose the distribution of the duration segment $f_\tau(x)$. Considering an extreme case when $\theta = 1$, the duration becomes $\mu + \tau_1$. The comparison with the original duration calculation suggests the distribution of τ_1 should also follow one exponential distribution. Hence, we set $f_\tau(x) = \lambda e^{-\lambda x}$.

With the chosen oracle function and duration segment distribution, we show that the probability density function of the duration in PoETA is

$$f_{\text{PoETA}}(x) = \theta\lambda e^{-\theta\lambda(x-\mu)} \tag{7}$$

By setting θ and λ such that $\theta\lambda = \frac{1}{\Lambda}$, the distributions of duration in PoET Equation 6 and PoETA Equation 7 become identical. Details can be found in Appendix B. Hence, the distribution of the duration in PoETA also satisfies the characteristics described in §III-C, and the second design goal is achieved.

3) **Trade-off between Security and Performance:** Now we target the third design goal. There is always a trade-off between security and performance. Particularly, we analyze the performance overhead and security with regards to different values of λ (the value of θ is then determined by $\theta = \frac{1}{\lambda\Lambda}$).

The performance overhead mainly comes from the extra invocations of `ecall_CreateWaitCertificate()`. Note that counter operations are the most costly (One counter write operation might take 80-250 ms [29]) in `ecall_CreateWaitTimer()` and `ecall_CreateWaitCertificate()`, here we use the frequency of counter operations to estimate the performance overhead (real evaluations are provided in §VII-B).

Consider a network of N nodes, the mean of the minimum duration $\mathcal{D}_\mathcal{M}$ is $\frac{\Lambda}{N}$. The number of counter operations for generating one block in PoET is roughly $N + 1$ (N `ecall_CreateWaitTimer()` for all nodes and one `ecall_CreateWaitCertificate()` for the leader node). In PoETA, the mean of intervals (duration segments) between two `ecall_CreateWaitCertificate()` invocations is $\frac{1}{\lambda}$. Hence, the number of counter operations for generating one block in PoETA is roughly $N(1 + \lambda\frac{\Lambda}{N})$ (one `ecall_CreateWaitTimer()` and $\lambda\frac{\Lambda}{N}$ `ecall_CreateWaitCertificate()` for each node). Hence, the performance overhead can be estimated as $\frac{\lambda\Lambda-1}{N+1}$. Note that $\lambda = \frac{1}{\theta\Lambda} \in [\frac{1}{\Lambda}, \infty)$. When λ is close to $\frac{1}{\Lambda}$, the performance overhead is close to 0, as PoETA becomes more like the original design. When λ approaches infinity, the performance overhead could grow really large. Counter operations will be performed constantly and frequently, and PoETA becomes more like PoW.

While λ affects the performance overhead, it also plays an important role in security. While the adversary could not preview the value of duration, she might alternatively try to infer whether the duration will be above a certain threshold and determine whether to skip the call to `ecall_CreateWaitCertificate()`. The adversary could learn it one step (duration segment) in advance since the duration accumulates step by step. Detailed analysis is in Appendix C. Simply put, larger λ suggests smaller steps the duration accumulates, forcing the adversary to wait until a time closer to the interested threshold, resulting in better security. For example, given that a network of 100 nodes with a localMean of 1000 seconds, if it is acceptable with the

```
WaitTimer{
    double requestTime
    double currentDurationSegment
    double accumulatedDuration
    byte[32] WaitCertId:sub:'n'
    double localMean
}
```

Fig. 8: `WaitTimer` Structure

mean of each step (duration segment) equal to 5 seconds, we have $\lambda = 0.2$ and overhead of $1.97\times$. If the security is more concerned and it is desired that the mean of each step is 1 second, we need to set $\lambda = 1$, resulting in an overhead of $9.89\times$. The real performance overhead under various settings is reported in §VII-B.

VII. Implementation and Evaluation

In this section, we present a prototype implementation of PoET and describe the evaluation results.

A. Implementation

We implemented PoET based on Hyperledger Sawtooth version 1.0.5, which is the most recent version that supports running Sawtooth PoET with SGX. The modules that are most relevant to PoET include BlockPublisher and ChainController. BlockPublisher is responsible for batch validation and inclusion in a block, and ChainController is responsible for block validation and fork resolution. These two modules correspond to the two roles of a validator, *i.e.*, the client and the server. We also implemented the Sawtooth patch mentioned in subsection V-A as it is not yet included in version 1.0.5.

When implementing PoET, we aim to minimize the modifications to the original PoET framework. Since the shortest duration is still the decider of a leader election process, we keep the block verification and fork resolution in ChainController unchanged. We also keep the original PoET block generation flow in BlockPublisher. In PoET, the block publish flow is changed from waiting for one duration to wait for multiple duration segments. In the consensus module, we rewrote the two main ECalls `ecall_CreateWaitTimer()` and `ecall_CreateWaitCertificate()`. Instead of generating one final duration, `ecall_CreateWaitTimer()` will generate the first duration segment plus the `MINIMUM_WAIT_TIME`. The major modifications were applied to `ecall_CreateWaitCertificate()` as shown in algorithm 1. The key implementation flow is shown in the following steps:

- In the first step, a validator requests the first duration segment τ_1 along with the `MINIMUM_WAIT_TIME` by calling `ecall_CreateWaitTimer()`. A data structure called `WaitTimer` is introduced to record necessary information about the duration accumulation. As shown in Figure 8, `WaitTimer` consists of 5 components. Particularly, `requestTime` records the time when the current timer is created; `currentDurationSegment` is the newly generated duration segment τ_s; `accumulatedDuration` is the accumulated duration so far including the current duration segment, *i.e.*, `MINIMUM_WAIT_TIME` $+\sum_{i=1}^{s}\tau_i$;
- In the second step, a validator keeps waiting until the amount of time equal to `currentDurationSegment` elapses, and invokes `CreateWaitCertificate()` afterward. `ecall_CreateWaitCertificate()` then checks if the `currentDurationSegment` is passed. If yes, it will invoke the oracle function $\Theta()$: it generates a random number within $(0, 1)$ and outputs 1 if the random number is smaller than θ, or 0 otherwise. If the oracle function outputs 1, a `WaitCertificate` will be created. Otherwise, another duration segment will be sampled, and the `WaitTimer` will be updated and returned. Note the random values and duration segments are generated deterministically from the previous duration segments, sealed key pairs, and the previous certificate.
- The third step is after a validator repeats the second step until a `WaitCertificate` is generated successfully. The validator then submits it to the network.

Algorithm 1: ecall_CreateWaitCertificate()

Data: `inSealedSignupData`, `inSealedSignupDataSize`, `inSerializedWaitTimer`, `inWaitTimerSignature`, `inBlockHash`, `inSerializedWaitCertificateLength`, `outWaitCertificateSignature`
Result: Generate `WaitCertificate` or update `WaitTimer`

```
variables initialization;
check re-entrance attacks;
if the timer is expired and within the predefined window then
    continue;
else
    abort;
if Θ() = 1 then
    generate WaitCertificate;
else
    sample another duration segment τ_s;
    update WaitTimer;
```

B. Evaluation

We evaluate PoET with regards to the three design goals. We first evaluate the consistency between the distributions of duration in PoET and PoET. We then evaluate the security and performance under different values of θ and discuss the trade-off between them.

1) **Consistency:** As analyzed in §VI-B2, when $\theta\lambda = \frac{1}{\Lambda}$, the distributions of duration in PoET and PoET are the same. We verified this by simulations. Particularly, we evaluated how different values of λ affect the distribution with localMean equal to 100 seconds. We simulated the generation of duration in PoET under 5 different values of λ. With each value of λ, we collected $1,000,000$ durations and plotted the normalized histogram as shown in Figure 9. We can see that the distributions of duration in PoET under different values of λ are

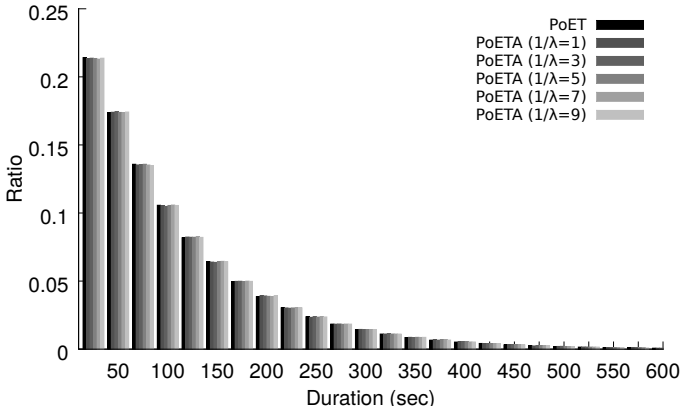


Fig. 9: Duration distributions of PoET and PoETA

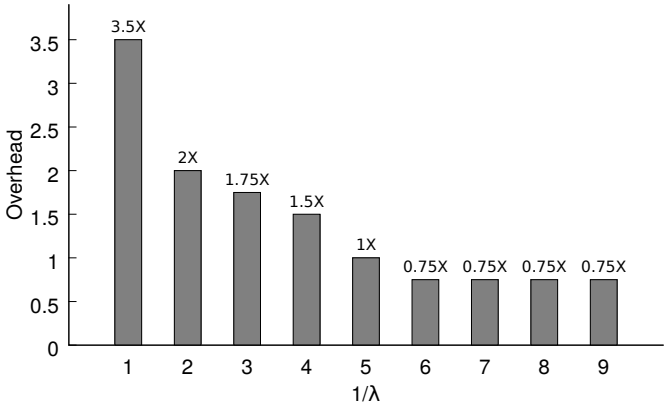


Fig. 11: Performance overhead under different λ values.

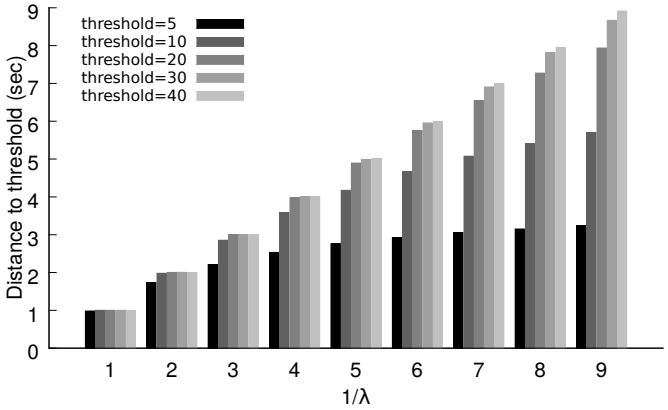


Fig. 10: Distances to thresholds under different values of λ. Larger λ suggests smaller steps the duration accumulates, forcing the adversary to wait until a time closer to the interested threshold, and thus resulting in better security.

almost the same as that of PoET. Hence, we claim that the consistency goal is achieved.

2) **Security and Performance:** We now detail the security and performance evaluations of PoETA. Both are evaluated under different values of λ. For the security evaluation, we measure how much time in advance when one could determine that the current `durations` would exceed certain thresholds (such as 10 seconds or 20 seconds). For the performance evaluation, we measured the CPU utilization of running our implemented PoETA. We experimented with 9 different values of λ.

Security. We tested 5 different thresholds (ranging from 5 seconds to 40 seconds) around the default target wait time (20 seconds). For each threshold, we simulated the duration generation process and collected those samples with durations larger than the threshold. For each sample, we recorded the accumulated durations right before they went beyond the threshold and calculated the mean of these accumulated durations. The results are averaged from $1,000,000$ samples each, as shown in Figure 10. We can see that the average distances from the thresholds when one realizes that the final duration would exceed the thresholds are less than the mean of duration segments, *i.e.*, $\frac{1}{\lambda}$. Hence, a larger λ provides better security.

Performance. We measured the CPU utilization when running PoET and PoETA. We used perf, a performance analyzing tool, to measure the CPU utilization of one validator during the period that 10 blocks are published in the network, which has an average time cost of around 10 minutes. The CPU utilization of PoET was 0.4%. For PoETA, we tested 9 different values of λ as in the security evaluation, and obtained the CPU utilization ranging from 1.8% to 0.7%, resulting in a overhead ranging from $3.5\times$ to $0.75\times$ as shown in Figure 11.

The evaluated performance overhead is smaller than that is analyzed in §VI-B3. The reason is that the overhead estimation is calculated only from counter operations, while the actual execution cost includes both enclave operations and non-enclave operations. For example, in the host application of Sawtooth, there is a loop checking whether a given duration (or duration segment for PoETA) has elapsed. The default frequency of such checks is 10 times per second.

Trade-off. By comparing the security and performance under different values of λ, we can see the trade-off, *e.g.*, better security leads to more considerable overhead. For PoETA systems requiring faster block generation rate, *i.e.*, smaller target wait time, better security guarantee is needed. Furthermore, even for the largest performance overhead we have evaluated, the CPU utilization is still quite low (1.8%), making PoETA a resource-efficient alternative of PoW. On the other hand, the trusted timers provided on existing SGX platforms have a granularity of one second. Hence, $\lambda = 1$ would be a reasonable choice for most scenarios if 1.8% of CPU utilization is acceptable.

VIII. Discussion

A. Eliminating Multi-Certificate Attacks.

To completely eliminate Multi-Certificate Attacks, the following approach can be taken: In the sealed data, we keep track of the previous Certificates. For each round, one node is allowed to generate only one duration and one Certificate. To enforce this rule, we can use the monotonic counter and a flag for tracking the previous Certificates, and any violation will terminate the current enclave instance and require re-authentication. Notably, a node is only allowed to generate a waitTimer if the previous Certificate is not used. The monotonic counter is leveraged to prevent the sealed data from the

rollback attacks. When a waitTimer is generated, the flag is set to true. Once the flag is true, no more waitTimers could be generated for the current round unless the previous round has another certificate with a smaller duration than that in the sealed data is found.

While this design could eliminate MCAs, it suffers from the following disadvantages: First, all Certificates must be stored in the sealed storage, introducing both significant storage and computation overhead that grow linearly with regards to the length of the blockchain and the number of nodes in the network. Though utilizing algorithms and data structures might help with the issues of storage size and the computational overhead, the overhead might still be high compared with PoET. Second, this option highly depends on the secure implementation of complex algorithms and data structures while evaluating the security of a larger TCB requires considerable effort. Third, when a node misses the chance to claim the waitTimer in time due to some unexpected long latency of creating Certificate, it will lose the chance to publish the current block since it is allowed to generate the duration only once for each round by design. Therefore, a solution like PoET is of significance.

B. Alternative Design

Instead of sampling a random duration segment, one alternative design is to use constant duration segments. However, two reasons prevent us from adopting the constant interval design in PoET: First, it comes with a distribution that is different from PoET. From our discussion with PoET designers, the distribution of PoET leads to desired blockchain properties (higher block processing rate, constant block generating rate, and fewer forks). Hence, it is preferred to preserve these properties by a design with an identical distribution. Second, with the constant interval design, it is more likely for two nodes to generate the same duration, leading to a network jam.

C. Potential Attacks to Other TEE Settings

As Hyperledger Sawtooth PoET is implemented on Intel SGX, our analysis focuses on SGX. But the attacks presented in section V are not specific to SGX. Other TEEs, such as AMD Secure Encrypted Virtualization (SEV), may also be vulnerable to similar attacks. The adversary controlling the virtual machine monitor may restart the VM to roll back the state of the PoET consensus protocol. However, as SEV's protected domain is the entire virtual machine, duration may not be exposed to the adversary who stays outside the VM. As such, we envision a secure implementation of PoET on AMD SEV remains an interesting research problem.

D. Multiple Malicious Nodes

Next, we discuss both parallel and sequential MCAs with just one malicious node. They could be performed with multiple nodes under the adversary's control. In such cases, the colluding nodes may gain more advantages during block leader election. For example, they can bypass Z-test by mining blocks in an alternating manner. Consequently, the group of malicious nodes could control the chain with fewer efforts. As discussed in §III-D, forks are resolved by selecting the one with larger aggregated localMean of all blocks on the chain. As such, if the malicious nodes can obtain a high probability of winning by generating blocks with a larger aggregated localMean, they can thereby create a winning fork, which is the well-known selfish mining attack.

IX. Related Work

A. Integrating SGX into Blockchain

Increasingly, SGX has been integrated into blockchain for various reasons, such as leveraging SGX to protect smart contracts, deploying SGX as part of blockchain consensus to provide security from hardware, and protecting the privacy of blockchain nodes in SGX.

PDO [13] leverages SGX to run smart contracts in SGX enclave to provide data confidentiality, execution integrity and also enforce data access policies. Inspired by PDO, Ekiden [16] aims at protecting smart contracts in SGX with the support of permissionless and open settings, while PDO targets permissioned and controlled settings. FastKitten [18] also leverages SGX to protect smart contracts but with a focus on off-chain execution of multi-round contracts.

BITE [30] protects the privacy of light clients in Bitcoin using SGX. Teechain [27] deploys SGX to protect treasuries for establishing off-chain payment channels. SCIFER [9] leverages SGX's remote attestation for identification in the permissionless network. Recently, FastBFT [28] proposes a new BFT protocol which deploys SGX for secret sharing.

B. Consensus Protocols in Blockchain

Proof-of-Work (PoW) [35] is the original Bitcoin consensus protocol. The fundamental goal of PoW was to address Sybil Attacks by requiring validators to solve a puzzle through non-negligible cost. The process of finding a solution is called mining. Validators can only participate in the distributed blockchain if they have proposed a correct solution. The critical drawback of PoW is its energy resource expense, transaction slowness, and double-spending. Consequently, to address the shortcomings of PoW, various new consensus protocols as discussed below have been proposed:

- **Proof-of-Ownership** [22] uses Bitcoin's decentralized ledger to verify that users with the private key associated with the signature can prove they are the owner. It allows artists or businesses to certify the integrity, date of publication, and ownership of creations and contracts. Proof of Ownership is always attached to a piece of data using cryptographic functions. This makes it impossible to alter the data after certification.
- **Proof-of-Capacity** [19] aims to reduce the amount of wasted energy and computing resources by utilizing hardware storage for blockchain production. Similar to the computation of PoW, storage space is used to construct a proof of validity, allowing a validator to propose a block to the consensus. Validator membership requires a validator to prove validity through storing and retrieving shards of a given file.
- **Proof-of-Luck** [32] is based on the use of SGX-enabled CPUs, similar to PoET. Proof of Luck utilizes SGX platform's random number generator to choose a consensus

leader for block proposal. This provides low latency transaction validation, deterministic confirmation time, negligible energy consumption, and equitably distributed mining.

- **Proof-of-Stake** (PoS) [25] is introduced to reduce resource consumption encountered with PoW. The core idea is that participants deposit a value of the stake, which will be lost if the validator does not conform to the protocol. Key advantages are energy savings and improved throughput. Hash puzzle-solving is replaced by stake selection. This protocol suffers from centralization governance and Nothing-at-Stake attacks.
- **Proof-of-activity** (PoA) [11] is a hybrid approach of PoW and PoS. In PoA, the mining process starts as a standard POW mining process utilizing computing power to solve the puzzle and create a new block. When a new block is found (mined), the system switches to PoS, with the newly found block containing only a header and the miner's reward address. Based on the header details, a new random group of validators from the blockchain network is selected who are required to validate or sign the new block. The more crypto coins a validator owns, the more chances he or she has for being selected as a signer.
- **Proof-of-Reputation** (PoR) [21] is an optimization for selecting valid validator members over the existing PoW validator selection process. Instead of using instantaneous mining power to select members, PoR considers the validator's mining integrity. This is a formula calculation based on the total amount of valid work a miner has contributed since the system has been active. Any malicious activity will lower the miner's reputation and its voting power. An adversary can only be successful after gaining a positive reputation, requiring costly investment over time and resulting in fewer attacks.
- **Elapsed Time** (ET) [12] is a simplified version of PoET and Proof of luck. The key difference between PoETA and ET is whether the time interval is *determined or not* before time elapsing. In ET, the time interval is determined at the beginning and revealed partially during each OCALL. However, since a large variety of side-channel attacks against SGX enclaves have been reported, the time interval may be leaked in advance if it is generated within the enclave memory before time elapses, making ET vulnerable to the MCA attacks. In contrast, PoETA eliminates such possibilities. The interval will not be determined before the last invocation and thus will not be leaked in advance.

X. Conclusion

We have presented a security study of Proof-of-Eclipsed-Time (PoET) and demonstrated a new type of attack against this consensus protocol, namely Multi-Certificate Attacks. Vulnerabilities that enable the parallel Multi-Certificate Attacks have been reported to the community and patched in the latest version of Sawtooth. For the sequential Multi-Certificate Attacks that are difficult to remove from the current implementation, we propose PoETA, a new design of PoET, which only reveals the duration of the waitTimer after the elapsed time to the untrusted software. We have implemented PoETA and tested it with three sets of experiments. Our evaluation results show that PoETA has optimal performance with stronger security and also the identical duration distribution as PoET.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd Pedro Moreno-Sanchez for their very helpful comments. Guoxing Chen was partially supported by the National Natural Science Foundation of China under Grant No. 62102254 and Shanghai Pujiang Program under Grant No. 21PJ1404900. Zhiqiang Lin was partially supported by NSF grant 1834213.

References

[1] "Btp," https://www.hyperledger.org/learn/publications/kubernetes-case-study.

[2] "Hyperledger sawtooth examples," https://sawtooth.hyperledger.org/examples/.

[3] "Hyperledger sawtooth project," https://www.hyperledger.org/use/sawtooth.

[4] "Sawtooth poet document," https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html.

[5] "Sawtooth poet sgx," https://sawtooth.hyperledger.org/docs/core/nightly/1-2/sysadmin_guide/configure_sgx.html.

[6] "Scantrust," https://www.hyperledger.org/learn/publications/scantrust-case-study.

[7] "Sawtooth patch," https://github.com/hyperledger/sawtooth-poet/pull/35/files, Jul. 2019.

[8] "Sawtooth poet z-test," https://sawtooth.hyperledger.org/docs/core/releases/1.0/architecture/poet.html#z-test, Jul. 2019.

[9] M. Ahmed and K. Kostiainen, "Identity aging: Efficient blockchain consensus," *arXiv preprint arXiv:1804.07391*, 2018.

[10] M. Bastiaan, "Preventing the 51%-attack: a stochastic analysis of two phase proof of work in bitcoin," in *Availab le at http://referaat. cs. utwente. nl/conference/22/paper/7473/preventingthe-51-attack-a-stochasticanalysis-oftwo-phase-proof-of-work-in-bitcoin. pdf*, 2015.

[11] I. Bentov, C. Lee, A. Mizrahi, and M. Rosenfeld, "Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract] y," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 3, pp. 34–37, 2014.

[12] M. Bowman, D. Das, A. Mandal, and H. Montgomery, "On elapsed time consensus protocols." *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 86, 2021.

[13] M. Bowman, A. Miele, M. Steiner, and B. Vavala, "Private data objects: an overview," *arXiv preprint arXiv:1807.05686*, 2018.

[14] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[15] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[16] 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)*. IEEE, 2019, pp. 185–200.

[17] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[18] 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)*, 2019, pp. 801–818.

[19] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak, "Proofs of space," in *Annual Cryptology Conference*. Springer, 2015, pp. 585–605.

[20] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *International conference on financial cryptography and data security*. Springer, 2014, pp. 436–454.

[21] F. Gai, B. Wang, W. Deng, and W. Peng, "Proof of reputation: A reputation-based consensus protocol for peer-to-peer network," in *International Conference on Database Systems for Advanced Applications*. Springer, 2018, pp. 666–681.

[22] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 491–500.

[23] Intel, "SDK for Intel Software Guard Extensions," 2020, https://software.intel.com/en-us/sgx/sdk.

[24] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel Software Guard Extensions: EPID Provisioning and Attestation Services," Intel, Tech. Rep, Tech. Rep., 2016.

[25] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference*. Springer, 2017, pp. 357–388.

[26] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 203–226.

[27] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch, "Teechain: a secure payment network with asynchronous blockchain access," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 63–79.

[28] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Scalable byzantine consensus via hardware-assisted secret sharing," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 139–151, 2018.

[29] 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)*, 2017, pp. 1289–1306.

[30] 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)*, 2019, pp. 783–800.

[31] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Tel-Aviv, Israel, 2013, pp. 1–8.

[32] 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*. ACM, 2016, p. 2.

[33] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *19th International Conference on Cryptographic Hardware and Embedded Systems - CHES 2017*, 2017, pp. 69–90. [Online]. Available: https://doi.org/10.1007/978-3-319-66787-4_4

[34] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Manubot, Tech. Rep., 2019.

[35] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[36] I. Stewart, "Proof of burn," *bitcoin. it*, 2012.

[37] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 991–1008.

[38] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 54–72.

[39] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.

[40] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on intel cpus via cache evictions," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 339–354.

[41] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic side channels for untrusted operating systems," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 640–656. [Online]. Available: https://doi.org/10.1109/SP.2015.45

APPENDIX A
THE WINNING PROBABILITY OF THE ADVERSARY

The generation of duration of node t in n-th round is given by

$$X_t^n = \underbrace{\underbrace{\texttt{MINIMUM\_WAIT\_TIME}}_{a} - \underbrace{\mathsf{localMean}_n}_{b_n} * \log(r)}_{a \text{ and } b \text{ are constant positive integers}} \quad r \in (0,1)$$

So we have

$$Pr\left(X_t^n \geq u\right) = Pr\left(r < e^{\left(\frac{a-u}{b_n}\right)}\right)$$

and

$$Pr\left(X_t^{n+1} \geq u\right) = Pr\left(r < e^{\left(\frac{a-u}{b_{n+1}}\right)}\right)$$

Since $A = min(X_1^n, ..., X_{T-1}^n)$, and $X_t^n, t = 1, \ldots, T-1$ are independent, we have

$$\begin{aligned} Pr\left(A \geq u\right) &= Pr\left(X_1^n \geq u, \ldots, X_{T-1}^n \geq u\right) \\ &= \prod_{t=1}^{T-1} Pr\left(X_t^n > u\right) \\ &= e^{\frac{(T-1)(a-u)}{b_n}} \end{aligned}$$

Similarly, we have

$$Pr\left(B \geq u\right) = e^{\frac{(T-1)(a-u)}{b_{n+1}}}$$

The probability density functions of A and B can be obtained as follows

$$\rho\left(A = u\right) = \frac{\mathrm{d}}{\mathrm{d}u}\Big(Pr\left(A \geq u\right)\Big) = \frac{e^{\frac{(T-1)(a-u)}{b_n}} * (T-1)}{b_n}$$

$$\rho\left(B = u\right) = \frac{\mathrm{d}}{\mathrm{d}u}\Big(Pr(B \geq u)\Big) = \frac{e^{\frac{(T-1)(a-u)}{b_{n+1}}} * (T-1)}{b_{n+1}}$$

Now, we can calculate $P[K]$ as follows

$$\begin{aligned} P[K] &= Pr\Big(KX_T^n < A + B | X_T^n < A\Big) \\ &= \frac{Pr\Big(KX_T^n < A + B, X_T^n < A\Big)}{Pr(X_T^n < A)} \end{aligned}$$

where $Pr(X_T^n < A) = \frac{1}{T}$ and

$$\begin{aligned}
&Pr\Big(KX_T^n < A + B, X_T^n < A\Big) \\
&= \int_u \int_v Pr\Big(X_T^n < \frac{u+v}{K}, X_T^n < u\Big) * \rho(A = u)\rho(B = v)\mathrm{d}u\mathrm{d}v \\
&= \int_a^\infty \int_{(K-1)u}^\infty Pr(X_T^n < u) * \rho(A = u)\rho(B = v)\mathrm{d}u\mathrm{d}v \\
&\quad + \int_a^\infty \int_a^{(K-1)u} Pr(X_T^n < \frac{u+v}{K}) * \rho(A = u)\rho(B = v)\mathrm{d}u\mathrm{d}v
\end{aligned}$$

Appendix B
The distribution of the duration in PoETA

Given that $\tau_i, i = 1, 2, 3, \ldots$ are independent and identically distributed random variables with probability density function $f_\tau(x) = \lambda e^{-\lambda x}$, we first show the probability density function of $\Delta_k = \sum_{i=1}^k \tau_i$ is

$$f_{\Delta_k}(x) = \lambda^k \frac{x^{k-1}}{(k-1)!} e^{-\lambda x} \tag{8}$$

by mathematical induction:

- *Base:* When $k = 1$, $\Delta_1 = \tau_1$, we have

$$f_{\Delta_1}(x) = f_{\tau_1}(x) = \lambda e^{-\lambda x} = \lambda^1 \frac{x^{1-1}}{(1-1)!} e^{-\lambda x}$$

- *Inductive step:* show that for any $k \geq 1$, if $f_{\Delta_k}(x)$ follows (8), $f_{\Delta_{k+1}}(x)$ also follows (8). Note that $\Delta_{k+1} = \Delta_k + \tau_{k+1}$,

$$\begin{aligned}
f_{\Delta_{k+1}}(x) &= \int_{-\infty}^{\infty} f_{\Delta_k}(y) f_{\tau_{k+1}}(x-y) dy \\
&= \int_0^x \lambda^k \frac{y^{k-1}}{(k-1)!} e^{-\lambda y} \lambda e^{-\lambda(x-y)} dy \\
&= \lambda^{k+1} e^{-\lambda x} \int_0^x \frac{y^{k-1}}{(k-1)!} dy \\
&= \lambda^{k+1} \frac{x^k}{k!} e^{-\lambda x}
\end{aligned}$$

Hence, (8) holds for all $k \geq 1$.

Note that the outputs of the oracle function invocations are also independent and identically distributed. The probability that the final duration consists of $k = s$ duration segments is

$$\begin{aligned}
Pr[k = s] &= Pr[\Theta(1) = 0, \ldots, \Theta(s-1) = 0, \Theta(s) = 1] \\
&= Pr[\Theta(s) = 1] \prod_{i=1}^{s-1} Pr[\Theta(i) = 0] \\
&= \theta * (1-\theta)^{s-1}
\end{aligned}$$

The cumulative distribution function of Δ is

$$\begin{aligned}
Pr[\Delta < x] &= Pr[\sum_{i=1}^k \tau_i < x] \\
&= \sum_{s=1}^\infty Pr[\sum_{i=1}^k \tau_i < x | k = s] Pr[k = s] \\
&= \sum_{s=1}^\infty Pr[\Delta_s < x] \theta * (1-\theta)^{s-1}
\end{aligned}$$

Now we have the probability density function of Δ

$$\begin{aligned}
f_\Delta(x) &= \sum_{s=1}^\infty f_{\Delta_s}(x) \theta * (1-\theta)^{s-1} \\
&= \sum_{s=1}^\infty \lambda^s \frac{x^{s-1}}{(s-1)!} e^{-\lambda x} \theta * (1-\theta)^{s-1} \\
&= \theta \lambda e^{-\lambda x} \sum_{s=1}^\infty \frac{[(1-\theta)\lambda x]^{s-1}}{(s-1)!} \\
&= \theta \lambda e^{-\lambda x} * e^{(1-\theta)\lambda x} \\
&= \theta \lambda e^{-\theta \lambda x}
\end{aligned}$$

Hence, the distribution of the duration in PoETA becomes

$$f_{\text{PoETA}}(x) = \theta \lambda e^{-\theta \lambda (x-\mu)}$$

.

Appendix C
The expectation of time before knowing duration will exceed a threshold

We are trying to capture the scenarios that one node has already waited k duration segments before reaching the threshold Γ and the $k+1$-th duration segment will cause the aggregated duration to exceed the threshold, *i.e.* $\mu + \Delta_k < \Gamma$ and $\mu + \Delta_k + \tau_{k+1} \geq \Gamma$. Let Ω denote such $\mu + \Delta_k$, which is the earliest time when one can determine that the duration will exceed the threshold. Since Δ_k and τ_{k+1} are independent, the joint probability distribution can be represented as

$$f_{\Delta_k, \tau_{k+1}}(x, t) = f_{\Delta_k}(x) * f_\tau(t)$$

The probability of having at least $s+1$ segments is

$$Pr[k \geq s+1] = \sum_{k=s+1}^\infty \theta * (1-\theta)^{k-1} = (1-\theta)^s$$

The expectation of $\Omega - \mu$ given the final duration will exceed Γ is

$$
\begin{aligned}
&E(\Omega-\mu)\\
=&\int_0^{\Gamma-\mu}\int_{\Gamma-\mu-x}^{\infty} x\frac{\sum_{s=1}^{\infty} f_{\Delta_s,\tau}(x,t)Pr[k>=s+1]}{Pr[\mathcal{D}>\Gamma]}dtdx\\
=&\int_0^{\Gamma-\mu}\int_{\Gamma-\mu-x}^{\infty} x\frac{\sum_{s=1}^{\infty} f_{\Delta_s}(x)f_{\tau}(t)Pr[k>=s+1]}{Pr[\mathcal{D}>\Gamma]}dtdx\\
=&\int_0^{\Gamma-\mu} xe^{\theta\lambda(\Gamma-\mu)}\sum_{s=1}^{\infty}\lambda^s\frac{x^{s-1}}{(s-1)!}e^{-\lambda x}(1-\theta)^s\int_{\Gamma-\mu-x}^{\infty}\lambda e^{-\lambda t}dtdx\\
=&\int_0^{\Gamma-\mu} xe^{\theta\lambda(\Gamma-\mu)}(1-\theta)\lambda e^{-\lambda x}e^{(1-\theta)\lambda x}e^{-\lambda(\Gamma-\mu-x)}dx\\
=&e^{-(1-\theta)\lambda(\Gamma-\mu)}\int_0^{\Gamma-\mu}(1-\theta)\lambda x e^{(1-\theta)\lambda x}dx\\
=&(\Gamma-\mu)-\frac{1-e^{-(1-\theta)\lambda(\Gamma-\mu)}}{(1-\theta)\lambda}
\end{aligned}
$$

Hence, The expectation of time before knowing duration will exceed the threshold Γ is

$$
\begin{aligned}
&E(\Gamma-\Omega)\\
=&\Gamma-\mu-E(\Omega-\mu)\\
=&\frac{1-e^{-(1-\theta)\lambda(\Gamma-\mu)}}{(1-\theta)\lambda}
\end{aligned}
$$

When $\theta\lambda=\frac{1}{\Lambda}$, the expectation can be written as

$$
\begin{aligned}
&E(\Gamma-\Omega)\\
=&\frac{1-e^{-(\lambda-\frac{1}{\Lambda})(\Gamma-\mu)}}{\lambda-\frac{1}{\Lambda}}
\end{aligned}
$$