Narrator: Secure and Practical State Continuity for Trusted Execution in the Cloud

Jianyu Niu[*]
Southern University of Science and Technology
niujy@sustech.edu.cn

Wei Peng[*]
Southern University of Science and Technology
pengw@mail.sustech.edu.cn

Xiaokuan Zhang
George Mason University
xzhang46@gmu.edu

Yinqian Zhang[*] [†]
Southern University of Science and Technology
yinqianz@acm.org

ABSTRACT

Public cloud platforms have leveraged Trusted Execution Environment (TEE) technology to provide confidential computing services. However, TEE-protected applications still suffer from *rollback* or *forking attacks*, in which their states could be rolled back to a stale version or be forked into multiple versions, resulting in *state continuity* violations. Existing solutions against these attacks either rely on weak threat models based on *centralized trust* (*e.g.*, trusted server) or suffer from large performance overheads (*e.g.*, tens of state updates per second). In this paper, we propose Narrator, a secure and practical system, (1) that relies on a blockchain (*i.e.*, decentralized trust) and TEEs, and (2) that provides high-performance state continuity protection like unlimited and fast state updates for applications in cloud TEEs. The intuition behind our design is simple. Our design uses the blockchain to initialize a distributed system of TEEs, laying down the decentralized trust base with a small interaction overhead, while the distributed system provides performant state continuity protection. Our distributed system adopts a customized version of the consistent broadcast protocol and leverages advanced techniques to make state updates processed with one round trip delay on average. We build a proof-of-concept of Narrator on Intel SGX (*i.e.*, a representative design of TEEs) and do extensive experiments to evaluate its performance. Our evaluation results show that in a LAN environment with 5 nodes, Narrator can support about 6k state updates per second, meanwhile keeping the latency as low as 3 − 8ms. The throughput is 30× larger than that in ROTE and 70× larger than using a TPM counter.

CCS CONCEPTS

• **Security and privacy → Security in hardware**.

KEYWORDS

Trusted Execution Environment, cloud, state continuity, blockchain

[*]Jianyu Niu, Peng Wei, and Yinqian Zhang are affiliated with the Research Institute of Trustworthy Autonomous Systems and the Department of Computer Science and Engineering of Southern University of Science and Technology (SUSTech).
[†]Corresponding author.

This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560639

ACM Reference Format:
Jianyu Niu, Wei Peng, Xiaokuan Zhang, and Yinqian Zhang. 2022. 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 '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560639

1 INTRODUCTION

Trusted Execution Environments (TEEs), *e.g.*, Intel Software Guard Extension (SGX) [31], ARM TrustZone [10] and AMD Secure Encrypted Virtualization (SEV) [51], enable applications to directly compute on confidential data without leaking secrets to an adversary who controls the computing infrastructures. For example, Intel SGX provides isolation for protected computation from untrusted operating systems (OS) by running codes and storing data in enclaves. Public cloud platforms such as Microsoft Azure [2], Amazon AWS [1] and Google Cloud [4] have already leveraged TEEs to provide confidential computing services, which aims to boost clients' confidence in their outsourced data or code.

However, stateful applications in TEEs that rely on the untrusted OS for persistent storage still suffer from *rollback* or *forking* attacks, in which their states could be rolled back to a correct but stale version or be forked into multiple versions, resulting in *state continuity* violation [15, 32, 40, 48, 52, 53, 58]. State continuity mandates that when a protected module resumes execution from an interruption (e.g., reboots, power outages, or system crashes), it should resume the same state before the interruption [48]. State continuity violation has severe consequences in many applications, such as payments [37], trusted storage [47], smart contract [33], as well as authentication rate limiting [53]. For example, in a payment system implemented in TEEs, an adversary can spend the same coins in multiple payments by reverting the states of its account balance (*i.e.*, double-spending attacks [41]). While there are prior works aiming to provide state continuity for TEEs [12, 40, 48, 52, 53, 55], they fall short in the following sense: First, hardware-based solutions like the SGX monotonic counters [7], Trusted Platform Module (TPM) counters [7], and NVRAM [48, 52, 53] are limited by the capacity of the underlying non-volatile storage, which limits the rate of write accesses to prevent worn out. For example, using a TPM counter for a state update takes about 97ms [53]. Moreover, such devices have a limited number of write operations in their lifecycle (*e.g.*, 0.3-1.4 million in NVRAM [53]). Therefore, these solutions cannot be applied in cloud settings where thousands of state updates have to be processed per second [39].

Second, software-based solutions offload the trusted counter to either a single trusted server [12, 55] or a collection of distributed servers [40]. However, the former merely migrates the trust from the cloud provider to yet another centralized entity, failing to address the problem from its root cause, and the latter relies on trusted administrators for system initialization [40], which still suffer from the issue of trusting a single party. Therefore, in this paper, we aim to answer the following question: How to design a system that provides unlimited and fast state continuity protection for clients' applications in cloud TEEs without relying on centralized trust?

An intuitive solution is to leverage blockchain [21, 41, 41, 42] as a *decentralized trust anchor*. Blockchain such as Bitcoin [41] and Ethereum [56] realizes an append-only log that does not rely on a central authority. Prior studies [16, 33] have used blockchain to record every state update of protected applications to prevent rollback and forking attacks. However, such a design is not practical. Each state update requires a transaction to be processed and confirmed by the blockchain, which is expensive in terms of long transaction latency (*e.g.*, up to one hour in Bitcoin and several minutes in Ethereum), low transaction throughput and high service fees for each transaction (*e.g.*, \$1.6 on average for one transaction in Bitcoin [3]).

We present Narrator, a system based on decentralized trust to provide performant state continuity protection for cloud TEEs. There are two key ideas behind our design. *First*, we keep the interaction with blockchain rare and outside the critical path of frequent state updates or reads. To this end, we only use an external blockchain to initialize a distributed system of TEEs, laying down the decentralized trust anchor (instead of recording frequent state updates). The use of the blockchain removes the reliance on a central trusted entity. *Second*, the distributed system initialized by the blockchain can work as a trusted system to provide state continuity protection for clients' TEE applications. The distributed system can be composed of TEEs located in the same data center since they have a short message delay (*e.g.*, RTT in about 1ms [6]). The short internode message delay enables the system to process requests quickly, which makes the whole system practical to meet the demands of cloud scenarios.

There are several challenges to realizing the above system. ***First***, the distributed system needs to tolerate unexpected failures and provide fast state updates with a small trusted computing base (TCB). To this end, Narrator adopts a customized version of the consistent broadcast protocol [19, 49] rather than complicated consensus protocols (*e.g.*, such as Paxos [35] or Raft [46]) for state updates. The insight behind our choice is that in Narrator, a protected application can decide the order of its requests without reaching an agreement with any other applications, however, consensus protocols have to ensure participants have the same global order of all requests. Therefore, it is not necessary to use complicated consensus protocols in Narrator. ***Second***, Narrator has to enable faulty nodes to resume their state and rejoin the system. In addition, the restart protocol has to ensure that an adversary cannot create parallel running systems to launch rollback and forking attacks. A secure restart protocol design is challenging. We have found an attack issue of the restart protocol in ROTE [40] (see Appendix A [43]). Narrator adopts a similar restart protocol as ROTE, but with a fix to the attack. ***Third***, without a trusted central entity, all TEEs have to complete the initialization process with the help of a blockchain in a distributed and automated manner. To make the initialization process efficient, we use a leader pattern, in which a TEE node is selected as the leader to coordinate with others. The key idea is that the leader can perform mutual attestation with other nodes to build a chain of trust.

We provide a formal security analysis of Narrator to show that TEE applications can use Narrator to protect their state continuity under a powerful adversary. We also implement Narrator on SGX and evaluated its performance in both LAN and WAN environments. Our evaluation shows that Narrator can support about 6k state updates per second, meanwhile keeping the latency as low as 3 – 8ms in a LAN environment with 5 nodes. The Narrator TCB increment is moderate (4300 LoC). While the proposed scheme is originally designed for Intel SGX, the method can be easily extended to different types of TEEs, *e.g.*, AMD SEV, and even between different types of TEEs, as long as they adopt similar systems.

Contributions. We make the following contributions:

- We design Narrator, a system to provide secure and practical state continuity protection for cloud TEEs. Our design creatively uses a blockchain to initialize a distributed system of TEEs to avoid expensive blockchain interactions.
- We propose a simple and efficient protocol for the distributed system based on consistent broadcast protocol. We also identify several performance bottlenecks and use tailored techniques to promote performance.
- We find the restart protocol of ROTE has a security issue, in which an adversary can successfully launch rollback and forking attacks. We propose a countermeasure to fix the security issue and adopt the fixed one in our system.
- We implement Narrator on SGX and have done experiments in both LAN and WAN environments to show its efficiency. Evaluation results show that Narrator can process thousands of state updates per second, and meanwhile has a low response time for state updates and reads of several milliseconds.

2 BACKGROUND

2.1 Intel Software Guard Extension

In this work, we build our system atop Intel Software Guard eXtensions (SGX) [31], which is a microarchitectural extension to Intel processors that provide shielded execution environments, called *enclaves*, for applications to protect their sensitive data from untrusted system software. An SGX application is divided into trusted and untrusted components, with the trusted components protected by the enclaves.

Enclave identity. When an enclave is created, the hash value of its initial code and data is calculated by hardware and used as the enclave identity (*i.e.*, MRENCLAVE). Additionally, each enclave is signed by its developer—dubbed Independent Software Vendor (ISV) by Intel—before release. The hash value of the public signature verification key is used as the enclave's sealing identity (*i.e.*, MRSIGNER).

Attestation. SGX provides two attestation mechanisms: local attestation (LA), by which a local enclave verifies another enclave, and remote attestation (RA), by which the remote user verifies that the enclave code is running on a legitimate Intel CPU with proper microcode version and the enclave identity is the same as expected. A successful RA will allow the user to trust the enclave environment as well as the integrity of the enclave code. Secrets can be provisioned into the enclaves once a secure channel is established using remote attestation.

Sealing. In SGX, enclaves can securely store data outside the protected memory by the sealing mechanism. Specifically, an enclave protects the integrity and confidentiality of data by encrypting it with a private key called the sealing key. The sealing key can be configured accessible to all enclaves with the same MRENCLAVE or with the same MRSIGNER.

2.2 Rollback and Forking Attacks

Stateful enclave programs have to seal and store their state data on the disk such that they can resume their state after unexpected interruptions (*e.g.*, system reboots or crashes). A powerful adversary that controls the OS or the hypervisor can schedule enclaves (*e.g.*, stopping, killing, or restarting) and provide enclave programs with stale versions of sealed data. These abilities enable the adversary to launch rollback and forking attacks to violate state continuity.

- *Rollback attack.* The adversary can restart an enclave program and replay previously sealed state data to it. The stale data can bypass decryption and integrity checks, and will roll the enclave program's state back to a previous one. Furthermore, when reading any state variable, the instance will return stale data.
- *Forking attack.* The adversary creates multiple instances of the same enclave program with forking states. In particular, the adversary simultaneously runs multiple instances from the same state, and then feeds each instance with different input data. As a result, the execution and final states of these enclaves would be different and consequently, these instances reach to different states. For example, in a limited password guessing program [53], an adversary runs multiple enclave instances and tries different passwords for each instance.

2.3 Revisiting Existing Rollback Preventions

Existing rollback prevention methods usually bind each sealed state data on the disk with a freshness tag recorded in a trusted subsystem (rather than directly storing whole state data in the subsystem). After reboots, the freshness tags enable the enclave program to check whether the state data retrieved from the OS is the latest.

2.3.1 *Counter vs. State Digest.*

There are two abstractions of the freshness tags: trusted monotonic counter or state digest.

Trusted monotonic counter. A trusted monotonic counter is a tamper-resistant counter whose value, once incremented, cannot be reverted to a previous value [50]. When a protected application updates its state, it has two operations: incrementing the counter and storing the state (and/or the input) with the counter value in the disk. The sequence of these two operations further determines two methods: inc-then-store [40], in which the counter is first incremented, and store-then-inc [39, 53], in which state persistence is first executed before incrementing the counter.

Table 1: The comparison with existing methods to protect state continuity. RP, FP and CT & H are short for rollback prevention, forking prevention, and centralized trust & hardware, respectively. The performance is measured by the processed state updates per second and is ranked into the low ($\approx$ 10), medium ($\approx$ 100) and high ($\approx$ 1000). The costs refer to fees paid for additional hardware or service fees and are ranked into the low (< \$100), medium (\$100 – \$10000) and high (> \$10000) for one million state updates.

Protocols	RP	FP	Liveness	Performance	CT & H	Cost
ROTE [40]*	✗	✗	✗	Medium	✓	Low
Ariadne [53]	✓	✗	✓	Low	✗	Low
ADAM-CS [39]	✗	✗	✓	High	✗	Low
Memoir [48]	✓	✗	✓	Low	✓	Medium
Blockchain [16, 33]	✓	✓	✓	Low	✗	High
NARRATOR	✓	✓	✓	Medium	✗	Low

* We find that in ROTE, an adversary can launch rollback and forking attacks through the restart protocol. Due to space constraints, we illustrate these attacks and fix them in Appendix A [43].

Trusted state digest. A trusted state digest is a tamper-resistant digest of application states recorded in the trusted subsystem. There are two explicit techniques to use state digests: *execute-then-record* and *record-then-execute*. Given a request, in the first technique, the request is first executed, and the updated state is recorded, whereas in the second one, the request together with the current state is recorded, and then the request is executed.

Summary. Although both abstractions can provide state freshness, the main difference is that the monotonic counter approach does not provide a unique binding between stored states on the disk with the counter value. In other words, there may exist multiple stored states with the same counter value by strategically killing/restarting enclaves. As a result, the inc-then-store method does not have liveness [40], while the store-then-inc method is vulnerable to the forking attack [39, 53]. Therefore, our design adopts the state digest abstraction.

2.3.2 *Hardware vs. Software.*

The above two abstractions can be realized in hardware-based, software-based, or hybrid subsystems.

Hardware. Hardware realizations include SGX monotonic counter [7][1], TPM monotonic counter [39], and TPM NVRAM [48, 52, 53]. These realizations have limited performances including slow write and read response time, and limited write cycles. For example, incrementing a TPM counter for a state update takes about 97ms, and reading a counter for a state check takes about 35ms [53]. Besides, the number of write cycles for TPM NVRAM (used to record state digest) or SGX counter is only several million times [40, 48]. To address the limited write cycles, Memoir [48] uses additional hardware accessory, *i.e.*, uninterruptible power source (UPS) [48] to reduce frequent writes.

Software. As said previously, software implementations [40, 55] rely on weak threat models based on centralized trust. A detailed comparison with ROTE is provided in §7. By contrast, the approach [16, 33] that uses blockchain has a strong threat model (*i.e.* decentralized trust), but suffers from poor performance and is economically infeasible (§1). The root cause is that blockchain relies

[1] In the latest version, SGX does not support monotonic counters anymore [39].

on Byzantine consensus protocols (*e.g.*, PBFT [20] or Nakamoto Consensus [41]) to make sure all nodes have the same transaction sequence, which usually has high communication overhead. In addition, in a decentralized system, participants are geographically located, which leads to a high message propagation delay and poor performance.

Hybrid system. ADAM-CS [39] is a hybrid system that provides virtual monotonic counters based on a set of distributed TPM counters. The hybrid mode enables ADAM-CS to support thousands of increments per second. However, ADAM-CS adopts the inc-then-store method, which has a vulnerability window (VW), *i.e.*, the time taken from the last stabilized states to the system crashing or shutting down [39]. In ADAM-CS, the VW is around 10 ms, during which the updated state can be rolled back.

Summary. Table 1 provides a brief summary of existing solutions. To prevent memory wear out, hardware solutions [48, 53] have limited performance for state updates. The hybrid system ADAM-CS [39] can process thousands of state updates per second but has a vulnerability window for rollback attacks. Existing software systems either rely on a weak threat model [40] or have poor performance [16, 33]. In a nutshell, none of these solutions can achieve practical and secure state continuity protection for cloud TEEs.

2.4 Blockchains

Cryptocurrencies such as Bitcoin [41] rely on a linked list of block structures, referred to as the blockchain. Blockchain systems leverage Byzantine consensus to enable a set of mutually distrusting nodes to reach an agreement on an ever-growing blockchain, which further serializes and confirms a list of transactions. There are two types of Byzantine consensus protocols. The first is Nakamoto Consensus (NC) and its variants [24, 25, 41], which are simple, tolerant of churn, and can support thousands of participating nodes. However, NC only provides probabilistic security, in which a transaction has the probability to be reverted due to forks [41]. The rollback of transactions would violate the state continuity of enclaves' states [16]. Besides, clients have to keep synchronized with blockchains to confirm transactions (*e.g.*, the longest chain rule [41]). Since enclaves' communication interfaces may be controlled by the adversary, the synchronization cannot be guaranteed, and enclaves cannot easily and securely confirm transactions [23, 33].

Another type is Byzantine Fault Tolerant (BFT) consensus such as PBFT [20], Tendermint [18] and HotStuff [57]. BFT protocols rely on a group of participants with publicly known identities (*e.g.*, public keys) that forms a committee to manage the system. The committee runs a multi-phase commit protocol (e.g., two-phase in PBFT [20]) to produce blocks. Each committed block has an associated publicly verifiable authenticator (i.e., a set of signatures from the committee members), that can be validated by anyone. Unlike NC, BFT protocols can provide strong security, *i.e.*, a committed block and the associated transactions cannot be reverted, as long as less than one-third of committee members are Byzantine. Note that despite the permissioned setting, BFT protocols can be used to build either permissioned blockchains [9] or permissionless blockchains [28]. In this work, we choose BFT-based blockchains because of their strong security.

Table 2: Summary of Notations.

Term	Description	Term	Description
P	Enclave program	S_i	Program state
I_i	Input request	O_i	State output
r_i	Execution randomness	$\mathcal{L}$	Blockchain
n	Number of SEs	ID	Blockchain identifier
f	Number of faulty SEs	SD_i	State digest
uid	SEs' unique platform identifier	(sk, pk)	Key pair

3 OVERVIEW

In this section, we present the problem statement, an overview of our design, and the threat models. Table 2 provides a list of commonly used variables and terms.

3.1 Problem Statement

The states of applications running in enclaves can be modelled by a *multi-step* interactive, probabilistic program P [33], which can be further abstracted as:

$$\mathrm{P}(\mathrm{S}_i, \mathrm{I}_i) \rightarrow (\mathrm{S}_{i+1}, \mathrm{O}_i).$$

Given state S_i, the enclave executes the request I_i from OS, and then enter the next state S_{i+1} with an output O_i. Here, the program must either complete the process of the request and then update its state, or not advance its state at all. To guarantee crash resilience (*i.e.*, liveness property), each state transition of P has to be deterministic [33, 48, 53]. The non-determinism factors of a program like time, random numbers, and multithreading can be converted to be deterministic by various techniques [13, 14, 33, 45] and modeled with the input I_i. The program P can well represent applications like authentication rate limiting [53], smart contract [33], and payment system [37].

Stateful programs should resume the same state before interruptions (*e.g.*, reboots or system crashes) [48]. The above guarantee is also referred to as state continuity [15, 32, 40, 48, 52, 53, 58], which can be further decomposed into *safety* and *liveness* properties defined below.

Definition 3.1 (Safety). A stateful enclave program P should never enter into a stale state or inconsistent state with the following guarantees:

- **Rollback prevention.** Suppose an enclave program P advances to state s_i at time t, no enclave instance of P will resume states s_j ($j < i$) after time t.
- **Forking prevention.** Suppose two instances of an enclave program P start from the same state s_{i-1} and then advance to states s_i and s'_i, respectively, then $s_i = s'_i$.

Definition 3.2 (Liveness). An enclave program can resume its state after unexpected failures, like power shutdown, in a non-adversarial setting.

We stress that because TEEs do not prevent denial-of-service attacks from the adversary, liveness can only be achieved without active attacks. For example, the adversary makes the majority of State Enclaves (SEs) crashed in our design (§5.1). Therefore, we restrict our definition of liveness to settings where compromising liveness is not the goal of attacks. Our aim is to preserve liveness

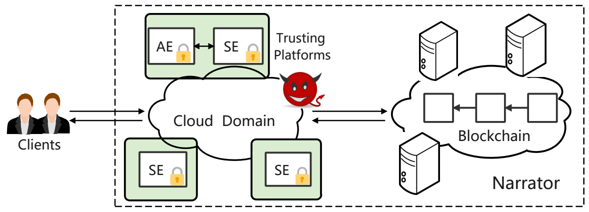


Figure 1: The architecture of NARRATOR.

when encountering unexpected system failures. In this work, we aim to achieve the following goals:

1) Security goals. We attempt to design a system to protect the state continuity, *i.e.*, *safety* and *liveness* properties, of cloud TEE programs, without relying on any trust assumption on centralized parties. Besides, our design should neither require any hardware changes nor (significantly) increase its attack surfaces.

2) Performance goals. We are particularly interested in providing practical state continuity protection for cloud TEEs. So our design should have low latency for state update and read operations (*e.g.*, in several ms), high throughput for processing enclave programs' requests (*e.g.*, thousands of state updates per second), and unlimited state updates.

3.2 Overview

Figure 1 illustrates the system architecture of NARRATOR. The core of NARRATOR is a performant distributed system, which contains $n = 2f + 1$ SEs running on different SGX-enabled platforms. Each SE can provide state continuity service to all the Application Enclaves (AEs) on the same platform. To tolerate unexpected failures, NARRATOR adopts a customized version of the consistent broadcast protocol [19, 49] rather than complicated consensus protocols (*e.g.*, such as Paxos [35] or Raft [46]) for state updates. The insight behind our choice is that in NARRATOR, an AE can decide the order of its requests without reaching an agreement with any other AEs, however, consensus protocols have to ensure participants to have the same global order of all requests. Therefore, it is not necessary to use complicated consensus protocols in NARRATOR. The simple protocol design also leads to a small TCB. Besides, NARRATOR also adopts several advanced techniques such as checkpointing, batch progressing, and pipelining for performance optimization.

To defend against rollback and forking attacks, the distributed system has to be securely initialized and configured based on some trust anchors. In NARRATOR, we use an external blockchain to initialize the system, which lays down the decentralized root of trust. Building a distributed system on top of a blockchain keeps the interaction with the blockchain rare and outside the critical path of state updates or reads, which makes the design performant and economically feasible.

3.3 System Model

We consider a distributed system of $n = 2f + 1$ SGX-enabled platforms on a cloud, a BFT-based blockchain $\mathcal{L}$, and a group of authenticated clients. The blockchain can provide a public append-only log for storing public data. Clients run applications on the SGX-enabled platforms, and each application operates one or more AEs. Each AE can access one SE on the same SGX-enabled platform, and so there are $n = 2f + 1$ SEs located on different platforms. An SE is considered to be faulty if it crashes down or does not complete the restart protocol after rebooting. We assume that at most f SEs are faulty at any time.

Threat model. Following related works [40, 48, 53], we consider a powerful adversary (*e.g.*, a malicious cloud administrator or an intruder) that can modify the system software stack (*i.e.*, OS or the hypervisor) on any SGX-enabled platforms, but cannot extract the memory contents or manipulate the running code in enclaves including both AEs and SEs. In particular, the adversary can schedule the execution of the enclaves including both AEs and SEs and replay their persistently stored data. The adversary cannot forge AEs' and SEs' messages since AEs build secure communication channels with the local SE (on the same SGX-enabled platform) using local attestation, and SEs build secure channels with each other using remote attestation. However, the adversary can eavesdrop, modify, and replay their messages, including the ones to and from the blockchain $\mathcal{L}$. Therefore, there is no assumption about the reliability of the communication network.

We assume that the attestation mechanism provided by SGX can guarantee that the outputs generated by the enclave are indeed from the code that is attested. Besides, we assume that the blockchain can provide a publicly-verifiable authenticator for committed transactions (§2.4), and the adversary cannot control the blockchains (*i.e.*, corrupting more than one-third of committee members) to violate its security property or forge authenticators for non-committed transactions. The adversary also cannot break standard cryptographic primitives. We do not consider Denial-of-Service (DoS) attacks on SEs, which could be launched by malicious AEs.

4 NARRATOR DESIGN

In this section, we present NARRATOR, a secure and practical system to protect the state continuity of cloud TEE programs. NARRATOR contains four important components: system initialization without using trusted central entity (§4.3), state update protocols (§4.4 and §4.5), state read protocol (§4.6), and AEs' and SEs' restart protocol (§4.7). In particular, state update protocol contains two designs: STAUP-BASIC (§4.4) and STAUP-OPT (§4.5). The latter is optimized to reduce the state update latency and improve the throughput at the price of a slightly larger TCB size.

4.1 State Continuity Technique

NARRATOR uses the state digest abstraction to maintain the enclaves' state continuity (§2.3). In particular, NARRATOR adopts the *record-then-execute* technique, in which the enclave first seals and stores $\langle \mathrm{S}_i, \mathrm{I}_i, r_i, \mathrm{SD}_{i-1} \rangle$ on the disk and *records* the hash SD_i in the trusted module, then *executes* the request, updates its state to S_{i+1} and reveals the output O_i. The state digest $\mathrm{SD}_i = \mathrm{H}(\mathrm{S}_i || \mathrm{I}_i || r_i)$ enables the enclave to recover states without compromising security. If the enclave crashes before the output is published, it can redo requests I_i and reveal the output O_i. Here, to ensure the enclave will produce the same output when redoing the requests after rebooting, the program should be deterministic.

NARRATOR does not adopt another technique, *execute-then-record*, because of its vulnerability to side-channel attacks [17, 54]. Specifically, an adversary may repetitively replay the input requests I_i

on the same state S_i and then observe side channels (*e.g.* the size of the encrypted state and output, and the time to perform the request) before allowing the state to be updated. For example, in a limited password guessing application, the adversary can try different passwords and then observe the timing side channel before it finds the right one. Although the side-channel attack and the associated countermeasures [29, 44] are not the focus of this work, we hope our design does not introduce new attack surfaces. Therefore, Narrator adopts *record-then-execute* technique. Besides, all sealed data and transmitted messages are kept with the same size (*e.g.*, using padding) such that an attacker cannot infer any information from the size.

4.2 Architecture

Figure 1 illustrates the Narrator architecture, which contains n Intel SGX-enabled platforms managed by a cloud provider, and an external blockchain $\mathcal{L}$. Each SGX-enabled platform runs multiple enclave instances, which can be divided into two types:

- **Application Enclave (AE).** AEs are enclave instances running applications created and uploaded by Independent Software Vendors (ISV). AEs handle clients' requests and return corresponding outputs.
- **State Enclave (SE).** SEs are enclave instances that run Narrator to provide state continuity service for AEs. Each AE can link to a local SE (also called the *target SE*) on the same platform and use Narrator library to protect its state continuity.

Due to space constraints, a detailed discussion of the above architecture and possible variants is provided in Appendix C.2 [43].

Narrator library provides two interfaces for AEs to protect their state continuity:

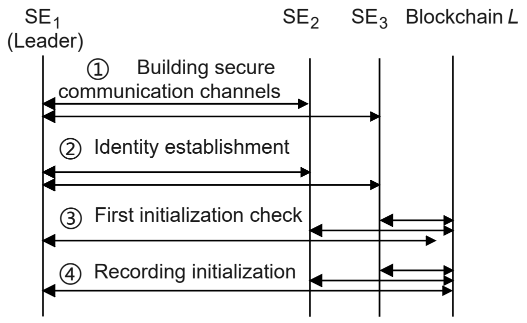
- writeState(*value*) $\rightarrow$ *ACK*: The interface enables an AE to write a new state digest to the Narrator system. If the operation succeeds, an *ACK* is returned to acknowledge the AE.
- readState() $\rightarrow$ *value*: The interface enables an AE to read its state digest from the Narrator system. Note that if the AE did not write any values previously, it gets an empty value (*Null*).

Given a state update, an AE first seals and stores a state snapshot on the disk and then calls writeState() to store associated state digests on Narrator. Once receiving an ACK, the AE can safely update its state and publish the output. When the AE needs to verify the freshness of data provided by OS after reboots, it calls the function readState() to obtain its latest state digest.

4.3 System Initialization

We aim to realize an initialization protocol, in which all SEs autonomously interact with a BFT-based blockchain to complete the configuration (without involving any trusted central party).

4.3.1 Blockchain Interfaces. All SEs can interact with a BFT-based blockchain $\mathcal{L}$ to complete the initialization process. The blockchain $\mathcal{L}$ further provides a key-value (KV) store abstraction [26]. These blockchains enable a client identified by ID to register a tuple $\langle key, blob\rangle$ to the blockchain, where *blob* is an arbitrary string that is mapped with the key. The identity ID of a client is referred to as the *blockchain identifier*, which can be generated from the public key of a digital signature scheme. When the tuple is first



SE1 (Leader)
SE2
SE3
Blockchain L
① Building secure communication channels
② Identity establishment
③ First initialization check
④ Recording initialization

Figure 2: The initiation architecture.

registered, a read-only entry with the identifier and the key is also created. Only the client identified by ID has the right to update the associated value with the key, whereas any party can read the value associated with the key. We model the interfaces of blockchain as:

- Blockchain.write(ID, $\langle key, blob\rangle$) $\rightarrow \sigma$: The interface enables a party with an identifier ID to post a string *blob* associated with the *key* on the blockchain. Once succeed, there is a returned authenticator σ over the request. Clients can run a public algorithm to verify whether the authenticator σ is authentically generated by the blockchain.

  Each write request contains a signature created by the client. For the first write of *key*, a read-only entry with the identifier *ID* and *key* is created. After the entry is created, the blockchain can check whether the client is properly authorized to perform the subsequent write operations (by verifying the associated signature).
- Blockchain.read(ID, *key*) $\rightarrow$ $(blob, \sigma)$: The read algorithm returns the *blob* associated with *key* and the authenticator σ. The σ enables the party to check the integrity of the return value.

For implementation, information of the blockchains (*e.g.*, genesis block) is hard-coded in SEs, which enables them to do backward verification of these authenticators. BFT-based blockchains like Algorand provide techniques to perform fast bootstrapping, which can accelerate the verifications [36]. Besides, in reality, the blockchain identifier can be merged with the *key*, by which a tuple $\langle key, blob\rangle$ represents an account, *key* is the identity (also known as an address [56]), and *blob* is the associated value.

4.3.2 Initialization Process. The initialization process should ensure that there is only one legitimate group of n SEs with known identities running on n different SGX-enabled platforms, *i.e.*, no forking groups. Without the above guarantee, AEs can link to more than one group, and each group can store a different state; the inconsistency of stored states will lead to a state continuity violation. Specifically, there are two challenges in the initialization process. *First*, all SEs have to build the PKI in a distributed way, which usually incurs a high message complexity. *Second*, to guarantee there is only one initialized group, a centralized system usually uses a trusted administrator to provide secret provisioning. In our design, without such a trusted party, an alternative way has to be proposed.

To address the first challenge, our design utilizes the leader pattern, in which an SE is delegated as the leader to coordinate other SEs. The leader can be delegated by the cloud provider, and the leader selection does not affect the security. The leader first performs mutual attestation with every other SE, by which they can attest to the identity of each other. After that, they build a chain of trust. Then, the leader can collect and distribute public keys from other SEs. To address the second challenge, our design relies on the interaction with the blockchain. During initialization, an SE has to post a record $\langle uid, blob\rangle$ on the blockchain, where *uid* is a unique identifier of the enclave program on the platform, and *blob* is the configuration information. When another SE on the same platform starts the initialization, it will generate the same *uid* and find the previous record in the blockchain, then aborts the initialization. As a result, no forking SE can be initialized in the same platform. Figure 2 illustrates the main procedure, which contains four steps.

❶ **Building secure communication channels.** The leader uses remote attestation to mutually attest to every other SE and meanwhile builds secure communication channels with all of them. In particular, when created, each SE (including the leader) calculates its enclave identity, *i.e.*, MRENCLAVE, and compares its identity with others' to verify whether the attested enclave is another SE [22].

❷ **Identity establishment.** All SEs (including the leader) generate an asymmetric key pair (pk, sk), and send their public keys to the leader. Then, the leader distributes the list of received public keys (denoted by $Cert$) to other SEs such that they can authenticate and build secure channels between each other.

❸ **First initialization check.** Each SE sends a read request to the blockchain via interface $\mathsf{Blockchain.read}(\mathrm{ID}, uid)$ (§2.4), where ID is the blockchain identifier of the SE (derived from its *pk*), and *uid* is the unique identifier of the SE on an SGX-enabled platform. SEs created on the same platform have the same *uid*. If returned *blob* is not null, the SE will abort the initialization since another SE on the same platform has been initialized. Otherwise, it executes the next step.

❹ **Recording initialization.** An SE sends a write request to the blockchain via the interface $\mathsf{Blockchain.write}$ $(\mathrm{ID}, \langle uid, \mathrm{H}(Cert)\rangle)$. When receiving a valid authenticator σ over the write request from the blockchain, the SE creates an SE state table containing state entries for other SEs and an empty AE state table, as shown in Figure 3. In particular, the SE puts the list of public keys pk_{SEi} to the SE configuration table and establishes pair-wise session keys key_{SEi} with all other SEs by running authenticated key agreement protocol. The SE sets the state digest for other SEs in the SE state table to *Null* when initializing. Next, the SE seals and stores its key pair, SE configuration table, and AE state table (see Figure 3). After that, the SE completes the initialization and can serve AEs' requests.

The identifier *uid* in the step ❸ can be realized by using the hash value of the seal key (binded with the SEs' identity MRENCLAVE), which can uniquely identify SEs on the same SGX-enabled platform. The read operation cannot guarantee to obtain the latest blockchain information, since the communication interface can be in the control of the adversary. However, this is not a concern, because an SE can only be initiated after a successful Blockchain.write(), which guarantees that the key of the SE has never been registered with the blockchain before.

More importantly, SEs should keep alive during step ❹. If the SE crashes before receiving *blob*, it cannot be initialized anymore. This is because normal crashes cannot be distinguished from malicious behaviors for creating forking groups. However, since the initialization process only happens once, the above failure is not a problem.

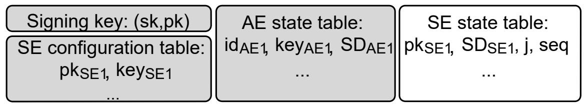


Figure 3: Layout of an SE's local memory. The data in the gray rectangle is sealed for state snapshot.

AE registration. When an AE starts to use Narrator, it performs the local attention on the target SE. The target SE's identity MRENCLAVE can be hard-coded to the AE implementation or provisioned by ISV. Then, AE can run an authenticated key establishment protocol with the target SE, by which they can share a key k. After that, SE creates an entry for the AE, which contains a unique identity of AE (*e.g.*, MRENCLAVE), the shared key, and an empty state digest (see AE state table in Figure 3).

4.4 State Update

When the target SE is ready to process AEs' state update requests, it follows the protocol shown in Figure 4. The protocol is based on Echo broadcast [49] and its variant [40], but we made some customizations to meet our design goals. A detailed discussion of the difference is provided in §7. AEs' and SEs' messages are transmitted in encrypted and authenticated channels using secret session keys. Besides, a nonce is added to each message to prevent the message replay attack. The detailed procedures are listed as follows:

❶ When receiving a request I_i, the AE first seals and stores a state snapshot $\langle \mathrm{S}_i, \mathrm{I}_i, \mathrm{SD}_{i-1}\rangle$ on the disk, where the state digest SD_{i-1} is $\mathrm{H}(\mathrm{S}_{i-1}||\mathrm{I}_{i-1})$. Then, AE calls the function $\mathsf{writeState}(\mathrm{SD}_i)$ to update its state digest. For clarity, we assume that needed randomness r_i is attached with I_i.

❷ When receiving the state update request, the target SE first caches it in a First-In-First-Out (FIFO) queue. Once the request is at the head and there are no other serving requests, the SE seals and stores its state snapshot $\langle \overline{\mathrm{S}}_j, \overline{\mathrm{I}}_j, \overline{\mathrm{SD}}_{j-1}\rangle$ on the disk, where $\overline{\mathrm{I}_j}$ is the AE's state digest SD_i.

❸ The SE sends prepare message $\langle \mathrm{Prepare}, \overline{\mathrm{SD}_j}, (j, seq)\rangle$ to all SEs (including itself) in the group. After reboot, an SE has to retrieve the latest state digests from other SEs (§4.7), so the index j is used to compare the freshness of state digests. Second, if a state update process is not completed (caused by unexpected failures), and later another request is provided, other SEs would store different $\overline{\mathrm{SD}_j}$. We use *seq* to distinguish the state digests with the same index, which could be caused by SEs' crash failures. After reboot, an SE obtains the latest state with the associated index and *seq*, it will increase *seq* by one and re-execute the state (§4.7).

❹ When receiving the prepare message, each SE updates the state digest of the target SE in the memory and sends back an Echo message that includes $\overline{\mathrm{SD}}_j$.

❺ After receiving more than $f + 1$ Echo messages, the target SE sends a decide message $\langle \mathrm{Decide}, \overline{\mathrm{SD}_j}, (j, seq)\rangle$ to all SEs in the group.

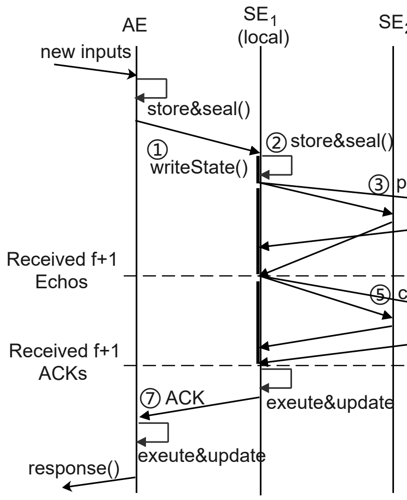


Figure 4: The overview of the state update protocol.

❻ When receiving the DECIDE message, an SE checks whether it has echoed the associated PREPARE message. If yes, it sends back an ACK that includes $\overline{\mathrm{SD}_j}$.

❼ When receiving more than $f+1$ ACKs, the target SE updates the AE's state digest and returns an ACK that includes SD_i. When receiving the ACK, the AE can safely update its state to S_{i+1} and publish associated output O_i.

4.5 Optimized State Update

The state update protocol in §4.4 is simple and can be implemented in a small TCB size, which is referred to as StaUp-Basic. Despite the simplicity, StaUp-Basic has several performance limitations. *First*, when processing a request, an AE has to seal and store a state snapshot in its local disk. For some applications like payments [37], the whole state is large, and so the seal and store operation incurs a heavy task for encrypting the data and IO writing. Evaluation results show that it takes several ms to seal and store 100KB data on the disk (§6). *Second*, an SE processes AEs' state updates in series. The serialization limits SE's capacity of processing requests per second (*i.e.*, throughput). In addition, when the number of supported AEs and associated requests are large, the low throughput will lead to a long service delay. *Third*, each update involves two rounds of message exchanges among SEs, which is the main factor impacting the latency and throughput.

To reduce the performance overhead, we propose a performant variant StaUp-Opt that can overcome the above limitations. StaUp-Opt adopts three optimizations: periodical checkpoint, batch processing, and pipelining. The periodical checkpoint enables AEs to store small requests rather than large states (optimizing step ❶ and ❷ in §4.4); the batch processing allows SEs to amortize the overhead for each state update (related with steps ❷ and ❼ in §4.4); and the pipelining structure makes a request only require one round of messages exchanges on expectation (related with step ❸-❼ in §4.4).

1) Periodical checkpoint. Our first optimization is to leverage checkpoints to reduce the size of sealed and stored data for AEs. For clarity, we describe the optimization in the view of AEs, which,

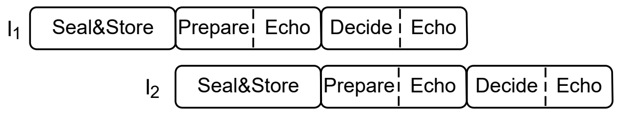


Figure 5: The pipelining structure in StaUp-Opt

however, also works for SEs. The key insight is that since the program is deterministic, the latest state can be recovered by executing all history inputs. This implies that AEs only need to seal and store input requests on disk and meanwhile record a summary of input history in SEs. For most applications, the input is usually much smaller than the whole application state.

However, the above solution requires AEs to store all history inputs (*i.e.*, large storage space) and to redo all requests in sequence to recover the latest state (*i.e.*, long recovery time) after rebooting. For these reasons, AEs periodically generate checkpoints (*i.e.*, state snapshots) and only store and redo inputs after the last snapshot for state recovery, which achieves a balance between these two methods. To this end, we have to modify procedure ❶ in §4.4. AE first stores the request (I_i, i) on disk. Then, it calls the function writeState(SD_i), where $\mathrm{SD}_i = \mathrm{H}(\mathrm{SD}_{i-1}||\mathrm{I}_i)$. The SD_0 is defined as $\perp$ when the system is initialized. For every k requests, the AE seals and stores an additional state snapshot $\langle \mathrm{S}_{jk}, \mathrm{SD}_{jk}\rangle$ $(j = 1, 2, ...)$ on the disk. When rebooted, an AE can retrieve the latest checkpoint and all requests after the checkpoint from OS, read its state digest in SEs, and securely recovers its state.

Storing input requests rather than states has another benefit. The sealed and stored data is not irreverent with the current state (except for the periodical checkpoint). Therefore, an SE can do this operation before the previous state update is done, which cuts off the operation time for state updates and further supports an efficient pipelining structure.

2) Batch processing. Our second optimization is to utilize batch processing to promote SEs' capacity for request processing. The idea behind our optimization is that each target SE has to serve multiple AEs, which run different applications. The state update requests between these AEs are independent, and so can be processed simultaneously. In other words, there is no need to serially process different AEs' state update requests.

In procedure ❷, an SE fetches a batch of requests (up to a maximum number m) from the queue and treats these requests as a single one. Then, SE follows the procedures ❸ - ❼ to process these requests. Once completed these procedures, the SE will process them one by one and send ACKs to corresponding AEs. The overhead of batch processing is negligible for two reasons. First, the increasing size of sealed and stored data only slightly increases the execution time, resulting in a slightly higher latency. However, this can be circumvented by the following pipelining technique. Second, network messages (*i.e.*, prepare and decide message) remain the same since the output size of the hash function is fixed no matter the size of inputs.

3) Pipelining. Our third optimization is the pipelining technique, as shown in Figure 5. In StaUp-Basic, there are mainly three stages for an SE to process a state update request: one stage for *seal&store* operation and two stages for message exchanges (illustrated in the bold line in Figure 3). Both of them are on the critical path of state updates. There are two observations behind our optimizations. First, the seal and store operation can be executed before the previous update requests are completed. The pre-processing can reduce the additional delay caused by the increasing size of sealing and storing data when batching requests. Second, the second-round message exchanges can be piggybacked to the first-round message exchanges.

We modify procedures ❸ - ❼ in STAUP-OPT. In particular, these five procedures are replaced by three new procedures.

❸ The SE sends a message $\langle \overline{\mathrm{SD}_j}, (j, seq), \mathrm{SD}_{decide} \rangle$ to other SEs in the group. If $\overline{\mathrm{SD}_{j-1}}$ is decided, then SD_{decide} is set null. Otherwise, SD_{decide} is $\overline{\mathrm{SD}_{j-1}}$.

❹ When receiving the state update proposal, an SE first stores the message. Then, if SD_{decide} is not null and the value has been echoed, it updates the state digest for the SE and returns an Echo message that contains both SD_j and SD_{decide}. Otherwise, it returns an Echo message that contains SD_j.

❺ If SD_{decide} is not null, and the target SE receives more than $f+1$ Echo messages that contains both SD_j and SD_{decide}, it updates the state digest SD_{decide} for the associated AE and then return the ACK response. If SD_{decide} is null, and the target SE receives more than $f+1$ Echo messages that contains SD_j, it will execute the procedure ❸, in which it sends a message $\langle \overline{\mathrm{SD}_{j+1}}, (j+1, seq), \overline{\mathrm{SD}_j} \rangle$ to other SEs in the group.

The pipelining structure enables NARRATOR to have a simple design (*i.e.*, unified message types) and higher efficiency (*i.e.*, one-round message exchange for requests on average). However, there is one limitation of using the pipelining structure. This is, when the target SE has no queued requests, the served requests are pending for new requests to complete the second round, resulting in a waiting latency. To address this issue, SEs can create a null request, which does not update any AEs' states, to complete the pending requests.

4.6 State Read

When an AE wants to check the freshness of its state or the sealed data from OS, it calls the function readState() to obtain the latest state digests from the target SE. When receiving the state read request, the target SE cannot directly return the recorded state digest of the AE. This is because there may be multiple local SE instances (*i.e.*, a forking attack), and the requested SE may not have the freshest one. Therefore, it has to check whether its state is the freshest. If yes, it can securely return the AE's state digest. The state read protocol contains four steps.

❶ An AE calls the readState() function to send the target SE a state read request.

❷ When receiving the request, the target SE sends a request to all SEs in the group to retrieve its latest state.

❸ Each SE checks its SE state table and sends back the stored state digest of the target SE.

❹ After receiving at least $f+1$ replies, the target SE first picks the state digest with the maximum index and sequence number. Then, the SE compares the state digest with the one generated by its state. If its state is the latest, it sends back the recorded state digest of the AE. Otherwise, it requires the sealed data from OS to resume its latest state.

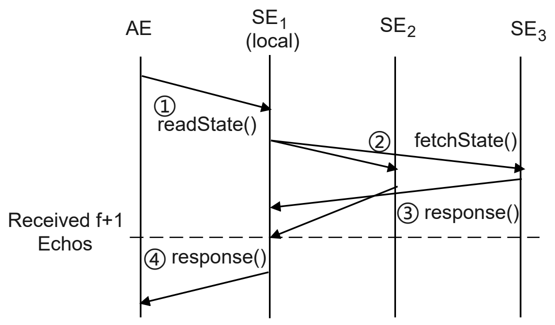


Figure 6: The overview of the state read protocol.

Due to rollback and forking attacks, an AE needs to check the freshness of its state when updating state (§4.7). However, this leads to extensive state read operations given a frequent state update. To address the high query overhead, we use a *post verification*, in which SEs check the freshness of AEs' requests. Given a state update, the AE sends the target SE both the current state digest and an updated state digest. If the AE's current state digest does not match the recorded one, the target SE returns an Error message, and then AE can read its latest state. Otherwise, the target SE also adds its current state digests into the state update request. When receiving the requests, other SEs do a similar check. If the target SE's current state digest is stale, they return an Error message, and then the target SE also sends the AE an Error message. Otherwise, the target SE will eventually return an ACK message for successful state updates.

4.7 Restart Protocol

We present the procedures that SEs and AEs run to recover their states after reboots.

1) SE Restart. The goal is to allow rebooting SEs to join the existing group, resume their states, and store other SEs' lasted state digests in memory. Our restart protocol adopts a similar session key update mechanism as the one in [40]. Specifically, an SE has to establish new session keys with other SEs since it has lost all previous ones. The session key works as an identity for an SE, which ensures only an SE on the same platform is working. However, we find that the restart protocol of ROTE [40] has a security issue, in which an adversary can utilize the protocol to create two parallel running systems and then launch rollback and forking attacks on TEE applications. Due to space constraints, we provide a detailed description and associated countermeasures in Appendix A [43]. The fixed restart protocol of SEs contains four steps.

❶ An SE first establishes new session keys with other SEs since it has lost all previous ones. Before session key establishment, other SEs have to check whether they are still in the group, *i.e.*, their session keys are still active. In particular, each SE pings other SEs, and they establish the new session key with the newly joined SE only if they receive the response from at least f SEs.

❷ The SE executes the steps ❷-❹ in the state read protocol (§4.6) to obtain its and other SEs' latest state digests. These can be combined in one read request.

❸ The SE picks the latest state digests with the maximum index and sequence number for each SE (including itself). For other SEs' state digest, it updates the SE state table with the picked value. For its state digest, it increases *seq* and re-executes the steps ❷-❻ in the state update protocol (§4.4) to update its latest state digest. These steps can guarantee the latest state digest is securely recorded in at least $f + 1$ SEs.

❹ The SE obtains the sealed state data from OS, and then compares it with the state digest from other SEs. If they match, the SE recovers its states and then becomes *non-faulty*. Only non-faulty SEs can process AEs' and other SEs' requests.

In step ❸, there may exist more than one state digests of an SE with the maximum index and sequence number. This case is triggered when the SE crashes and recovers several times without advancing to any new states. To ensure safety, under the above case, the SE stops recovering its state, which may affect the liveness property. However, this case rarely happens in a non-adversarial setting, and can be partially avoided by requiring a reboot to advance the SE's states.

2) AE Restart. After reboot, an AE has to recover its latest state. There are three steps: 1) the AE executes state read protocol to request the latest state digest from the target SE; 2) the AE obtains the latest sealed state from OS and then checks the freshness of unsealed state data, and 3) If these two match, the AE will update its state to the latest one and publish associated outputs.

5 SECURITY ANALYSIS

5.1 Liveness Analysis

The liveness property states that an AE can resume its state after unexpected failures like the power shutdown. Since the AE relies on Narrator for secure state recovery, we first have an observation of Narrator's f-liveness property.

Observation 1 (f-liveness). *An AE cannot read or write its state digest anymore when more than f SEs are faulty at the same time.*

The reason for the above observation is that given a state update or read request from an AE, the target SE has to obtain reply messages (*e.g.*, Echo or ACK messages in state read protocol in §4.4) from at least $f+1$ non-faulty SEs (including itself). Thus, more than f faulty SEs would make the AE's requests failed. Besides, a faulty SE has to complete the restart protocol (which involves the state read from at least $f+1$ non-faulty SEs) to enter the non-faulty status. Similarly, if more than f SEs are faulty, the SE cannot complete the restart protocol and become non-faulty.

The Narrator's f-liveness property provides a *detectable* security. This is, if an adversary violates this property by crashing more than f SEs and making the system down, it can violate the liveness property. However, the attack is easy to be detected by clients, and it also does not violate the safety property (since no AEs can update states or resume states after reboots). Therefore, we do not consider this an attack. Instead, cloud providers should choose suitable parameters for f such that the system can tolerate enough unexpected failures. With this property, we have the following theorem for AEs' liveness property.

Theorem 5.1 (Liveness). *With the f-liveness guarantee of Narrator, any AE can resume its state from unexpected failures in a non-adversarial setting.*

The proof is straightforward. For each state update, an AE has stored a sealed state snapshot on the disk and also the associated state digest in the Narrator system, which enables it to recover states from unexpected failures. Due to space constraints, we leave the detailed proofs in Appendix B.1 [43].

5.2 Safety Analysis

Without Narrator's f-liveness guarantee, AEs cannot update their states, in which rollback and forking attacks are impossible. Therefore, our safety analysis focuses on the existence of Narrator's f-liveness property.

Proof sketch. The proof has five main parts. First, we have the Observation 2, which says that the initialization process and restart protocol ensures that there is only one group of Narrator system for AEs. Second, we have Lemma 5.2, which says that SEs on the same platform can prevent the rollback attack. Third, with Lemma 5.2, we can easily prove that AEs of the same program are resilient to the rollback attack in Lemma 5.3 and forking attack in Lemma 5.4. Finally, we have the safety property of AEs in Theorem 5.5. We first introduce the observation of Narrator's uniqueness property.

Observation 2 (Uniqueness). *AEs of the same program P can link to only one Narrator system for state continuity services.*

The uniqueness of Narrator system is guaranteed by two procedures: the initialization process and the restart protocol. ***First***, the initialization process ensures that only one group of SEs is configured and serves AEs' requests. In particular, to complete the initialization, an SE has to append the configuration data with its unique identifier on the blockchain (step ❹ in §4.3). Later, when another SE is initialized and queries the same unique identifier from the external blockchain (step ❸ in §4.3), it will find the associated configuration data and know the previously initialized SE on the same platform. As a result, it will abort the initialization.

Second, the restart protocol guarantees that at most one SE on a platform is running, *i.e.*, communicating with other SEs. After rebooting, an SE has to refresh its session key with other SEs, and meanwhile receives replies from the majority of active SEs. If an SE completes the restart protocol and joins the Narrator system, other SEs on the same platform will have outdated session keys and cannot communicate with other SEs anymore. Therefore, AEs of the same program cannot link with multiple active SEs on the same platform. Next, we prove that SEs can prevent rollback attacks.

Lemma 5.2 (Rollback Prevention of SE). *Suppose an SE advances to state s_j at time t, no SEs on the same platform will resume states s_j $(j < i)$ after time t.*

This lemma says that an adversary cannot roll SEs' states back to a stale version. Due to space constraints, we leave the detailed proofs in Appendix B.2 [43]. Next, we can prove the rollback and forking prevention properties of AEs.

Lemma 5.3 (Rollback Prevention of AE). *Suppose an AE of the program P advances to state s_i at time t, no AEs of the same program will resume states s_j $(j < i)$ after time t.*

Proof. Given a successful state update s_i of the AE, there is an associated state update on the target SE, in which the associated state digest SD_i of the AE is recorded, Lemma 5.3 further ensures that once the state update of the SE is done, the SE will never roll its state back. Therefore, the AE of the same program cannot retrieve any state digest SD_j ($j \leq i$) from the target SE and also cannot resume states s_j ($j \leq i$). □

Lemma 5.4 (Forking Prevention of AE.). *Suppose two AEs of the same program P start from the same state s_{i-1} and then and advance to states s_i and s_i', respectively, then $s_i = s_i'$.*

Proof. The uniqueness property guarantees that AEs of the same program can only link to only one Narrator system. Without loss of generality, we assume that an AE of the program P first advances from state s_{i-1} to state s_i at the time t through an active SE on the same platform. By Lemma 5.2, another AE of the same program with state s_{i-1} cannot proceed to state s_i' because s_{i-1} is not the latest one, and SEs refuse to serve this state update. If the AE is rebooted, it will resume state s_j ($j \geq i$) and cannot enter state $s_i' \neq s_i$, too. □

Theorem 5.5 (Safety). *A stateful enclave program P should never enter into a stale state or inconsistent state.*

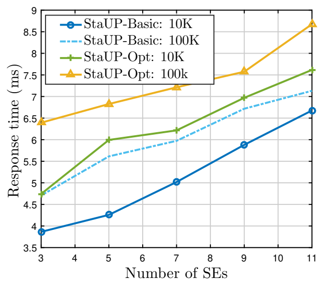
Proof. With Lemma 5.3 and Lemma 5.4, we have the safety property of enclave program P. □

6 PERFORMANCE EVALUATION

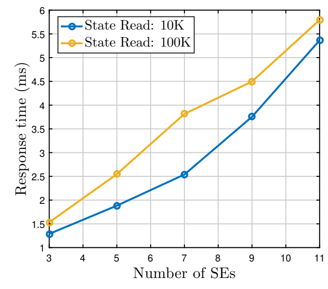
In this section, we evaluate Narrator to determine how it performs as compared to existing solutions. We consider two performance metrics: state write/read latency (§6.2) and throughput (§6.3). We compare Narrator with several designs: no-rollback protection, TPM monotonic counter, and ROTE [40]. We show that Narrator can provide both fast state write/read and high throughput, which could meet the performance requirements for cloud applications.

Implementation. We implemented Narrator on the Open Enclave SDK[2]. We used OpenSSL library for secure communication and cryptographic operations, such as the generation of random keys, encryption, and decryption. We use asymmetric cryptography for signing (ECDSA) and encryption (256-bit ECC). Besides, we use 128-bit AES-GCM in encrypt-then-MAC mode for symmetric message encryption and authentication. We implemented the SE in about 1500 LoC, Narrator library in about 2800 LoC, and a simple AE in about 800 LoC[3]. Since the interaction with the blockchain is out of the critical path of the state update and read, we do not implement it and evaluate the associated performance (*e.g.*, delay). We leave the evaluation as one of the future work. We use the ECDSA-based Intel DCAP attestation service for remote attestation of the SEs [5].

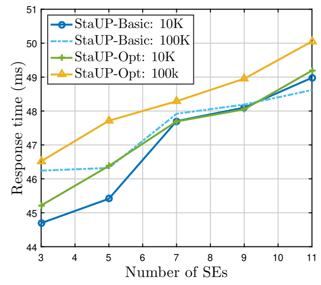
Experimental settings. We deployed Narrator in four Dell XPS-8940 desktops equipped with 32 GB RAM, an 8-core CPU (Intel I7-10700 @ 2.90GHz), and 512 GB SSD. We run Ubuntu 20.04 with Linux kernel 5.8.0 on these desktops. The four machines are connected with a 1Gbps wired local area network. We use NetEm [30] to simulate a LAN environment with 1.07 ± 0.03ms inter-SE RTT



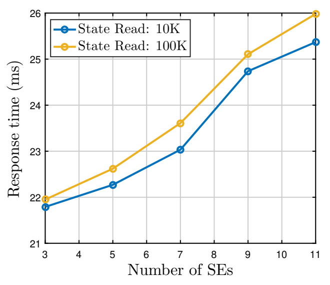
(a) State write response time in LAN.



(b) State read response time in LAN.



(c) State write response time in WAN.



(d) State read response time in WAN.

Figure 7: The state write/read response time with different number of SEs and AEs' state size (*i.e.*, 10KB and 100KB) in both LAN and WAN environments.

and a WAN environment with 20 ± 0.1ms inter-SE RTT. In particular, we run SEs on three different machines for the experiment of three SEs, while running more than one SEs on a machine if the number of SEs is larger than four.

6.1 Baselines

We consider three designs for comparison:

- **No-rollback protection.** For each state update, an enclave seals and stores a state snapshot on the disk. After reboot, the enclave fetches the latest sealed data from OS for state recovery. This design is used to illustrate the overhead introduced by rollback prevention methods.
- **TPM monotonic counter.** Given a state update, an enclave first increments its counter and then seals and stores a state snapshot with the counter value on the disk. To check state freshness, the enclave first reads the counter value and then compares the value with the one in the sealed data obtained from OS. This inc-then-store method does not provide liveness (§2.3).
- **ROTE.** The ROTE system also uses the above inc-then-store method, but provides a virtual monotonic counter by a distributed system [40].

6.2 State Write/Read Response Time

We evaluated the response time of state write or read operations. The response time is the total elapsed time from when a request is

[2]https://github.com/openenclave/openenclave
[3]The code can be found at https://github.com/pw0rld/Narrator.

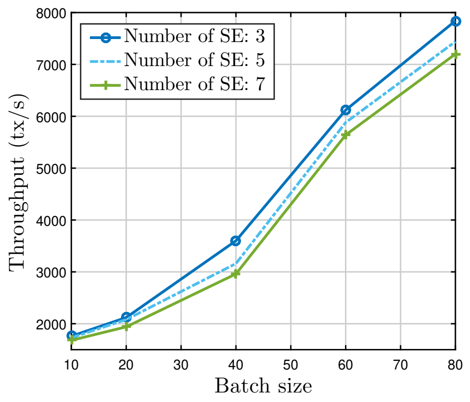
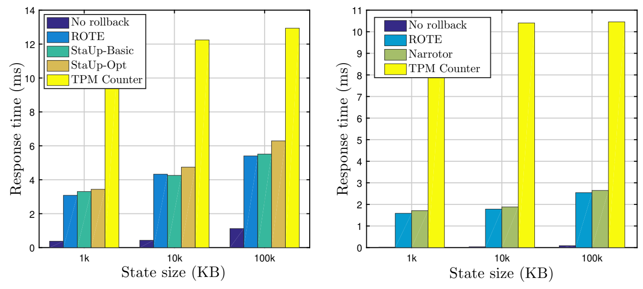


Figure 8: The Throughput of StaUP-Opt with different batch size and number of SEs given AEs' state size of 10KB.



(a) State write latency. **(b) State read latency**

Figure 9: The state write/read latency of different solutions with the varied AEs' state size.

made by an AE to the time an associated ACK is received by the AE. The batch size in StaUp-Opt is set to 50. (In §6.3, the impact of varied batch size on the throughput of StaUp-Opt is studied.) In particular, we do not consider the waiting time of service in the following measurements, which help us to know the optimal response time. Two environments, LAN and WAN, are considered.

LAN environment. Figure 7a shows the response time of state write with different numbers of SEs. We consider two settings of AE's state size: 10KB and 100KB. First, with the increase of SEs, the response time also increases. This is as expected, since the target SE needs to communicate with more SEs to collect their replies. Given AEs' state size 10KB, the response time is 3.87ms and 5.88ms in groups of 3 and 11 SEs, respectively. In the same setting, the response time in StaUP-Opt is larger than that in StaUP-Basic because of the additional processing time for other requests in the batch. Figure 7b shows the response time of state read with different numbers of SEs. Similarly, the increasing number of SEs also leads to a larger response time. When AEs' state size is 10KB, the delay is 1.29ms and 5.36ms in groups of 3 and 11 SEs, respectively.

WAN environment. Figure 7c and Figure 7d shows the response time of state write and state read in a WAN environment, respectively. Similar to that in LAN, the increase of SEs leads to a larger response time. In addition, results show that the message delay between SEs is a dominant factor in response time in a WAN network. Therefore, cloud providers should choose SEs located in the same data center to provide performant state continuity service.

6.3 Throughput

Figure 8 shows the throughput of StaUp-Opt protocol with different batch size in a LAN environment. The throughput is measured by the number of AEs' state update requests completed by SEs in one second. We do not evaluate StaUp-Opt because it proceeds requests in series, and its throughput can be easily derived by the associated response time in §6.2. The results show that the increase in batch size will significantly increase the throughput. For example, given a group of 5 SEs, the throughput is 1730 when the batch size is 10, whereas the throughput is 5880 when the batch size is 60. Besides, increasing batch size does not bring additional message overhead (§4.5). This implies that we can further increase the batch size for a larger throughput. Note that increasing batch size will slightly increase the response times, as shown in §6.2.

6.4 Performance Comparison

Latency. Figure 9a illustrates the response times of state write for different solutions. We consider a group of five nodes for both ROTE and Narrator. The results show that sealing and storing state snapshots without rollback protection can be done quickly within hundreds of microseconds. Given a state size 10KB, it takes about 429μs. The results also show that StaUp-Basic protocol in Narrator has a similar response time with ROTE due to the similar two-round write process. StaUp-Opt protocol has a slightly higher response time than them due to batch processing. Compared with Narrator and ROTE, the TPM counter-based solution has a much higher response time. Figure 9a shows the response times of state read for different solutions. We can obtain similar conclusions as that for state update.

Throughput. Both ROTE and TPM counter have to serially increment the counter value. Therefore, with the response time of their state update operations, we can compute their throughput. When the group size is 5 and state size is 10KB, the response times of ROTE and TPM counter-based solution are 4.32ms and 12.24ms, respectively. Thus, their throughput is 231 tx/s and 81 tx/s. Because of the similar response times, StaUp-Basic protocol has a throughput of 231 tx/s, whereas StaUp-Opt protocol has a throughput of 5880 tx/s when the batch size is 60 (§6.3). As we can see, the throughput of StaUp-Opt is 30× larger than ROTE and 70× larger than the TPM counter.

7 COMPARISON TO CONSISTENT BROADCAST PROTOCOLS AND ROTE

We discuss the comparison between consistent broadcast protocols [19, 49], ROTE [40], and Narrator. Narrator and ROTE are built based on consistent broadcast protocols, but adopt a two-round message change protocol (rather than one-round in the consistent broadcast protocols). The root cause is that Narrator and ROTE have to consider the recovery of participants, and the possible rollback attacks through the recovery mechanism. Besides, due to the integrity property of TEEs, a participant cannot lie about the sending messages, and so Narrator and ROTE can tolerate the majority (rather than one-third in [19, 49]) of faulty participants.

Narrator differs from ROTE [40] in four aspects. First, ROTE leverages the inc-then-store method with the monotonic counter abstraction, whereas Narrator uses the store-then-execute method with the state digest abstraction (§2.3.1). As a result, to guarantee safety, ROTE does not have liveness, *i.e.*, a crashed AE may not recover its state, whereas Narrator can achieve both safety and liveness properties. Besides, the above difference also leads to different sub-protocols, such as the state update and restart protocols. For example, in Narrator, an SE has to re-execute the latest state update after reboots to prevent rollback attacks. Second, ROTE relies on a centralized trust (*i.e.*, a trusted administrator to initialize the system), whereas Narrator is based on the decentralized trust (*i.e.*, using blockchains). Third, Narrator adopts a restart protocol with a fixed security issue in ROTE. We also note that Narrator adopts several techniques including the periodical checkpoint, batch processing, and pipelining structure to improve the state update performance. However, since batch processing and periodical checkpoint are general optimization methods, they can be directly applied to ROTE. Besides, the pipelining structure is closely related with the specific protocol, so it needs a tailored design to pipeline the two-round message change protocol in ROTE.

Due to space constraints, we also discuss the reconfiguration and hardware TCB update in Appendix C.1 and migration in Appendix C.3 [43].

8 RELATED WORK

8.1 Rollback Prevention

Non-volatile storage. There are many solutions based on the non-volatile storage, such as SGX counter [7] and TPM's counter and NVRAM [39, 48, 52, 53]. In [48], Parno *et al.* proposes Memoir, which is based on TPM NVRAM. Besides, an optimized variant called Memoir-Opt leverages the Uninterrupted Power Supply (UPS) and PCRs to address the limited write cycles of NVRAM. Similarly, Strackx *et al.* in [52] proposes ICE, in which contents of on-chip dedicated registers are written to persistent memory with the help of a capacitor at system shutdown. However, the additional hardware accessories are still under the control of the cloud provider and also leads to additional cost. Besides, these solutions have a long response time for state updates or reads. Unlike hardware modifications in Memoir and ICE, Ariadne [53] uses balanced Gray codes to realize a single bit flip when incrementing a counter, which can minimize the TPM NVRAM wear. However, Ariadne also has limited performance for using NVRAM. All these solutions rely on additional TPM PCR to detect forking instances.

Matetic *et al.* in [40] proposes ROTE, which realizes a distributed system to serve as the virtual counter for enclaves. Compared with the hardware counter, ROTE can process hundreds of state updates per second and has no limited write cycles. However, the ROTE system relies on a trusted party to initialize the system. Besides, ROTE cannot guarantee liveness for using the inc-then-store method. ADAM-CS [39] is a hybrid system that realizes a virtual monotonic counter based on a set of distributed TPM counters. ADAM-CS can support thousands of increments per second. However, ADAM-CS suffers from the vulnerability window, during which an adversary can revert states (§2.3).

Trusted third-party. Karapanos *et al.* in [34] proposes Verena, a web application platform in which a trusted server can provide integrity queries of a web page. In [55], a server with a trusted timestamping device is designed to provide trusted storage for clients. However, these solutions simply move the trust of a local OS to a remote server, which may also suffer from the attack. Besides, the trusted server becomes the attacked target and is also vulnerable to a single point of failure.

Blockchain and consensus protocols. Kaptchuk *et al.* in [33] leverages blockchain such as Bitcoin [41] and Ethereum [56] to maintain states for TEEs. By recording states in an append-only blockchain, enclaves perform secure computation without rollback attacks. However, as analyzed in §2.3, the performance of this method is poor because of the frequent interactions with performance-limited blockchains. Blockchains rely on Byzantine consensus to make all parties have the same transaction sequence, which usually has high latency for several rounds of communication, incurs high message complexity and requires less than one third of faulty nodes. What is more, each interaction with blockchains has to pay some fees for the provided service. making frequent state updates expensive.

Many studies [11, 27, 38] show that combining TEEs with Crash Fault Tolerant (CFT) consensus (*e.g.* Paxos [35] and Raft [46]) can securely work in the presence of the minority of Byzantine nodes. For example, Signal integrates Raft into SGX for securely storing clients' value [8]. However, CFT protocols like Raft are more complex than the adopted consistent broadcast protocol in Narrator. The additional complexity comes from the leader election and faulty leader detection, which unnecessarily increases the TCB size. Moreover, in Narrator, SEs can update their states in parallel, whereas Raft assigns a single node, called the leader, to handle all updates. The parallel processing makes Narrator more efficient. Therefore, Narrator is built on a customized consistent broadcast protocol, rather than CFT protocols.

8.2 Forking Prevention

TEE applications can use a hardware-based solution like Configuration Registers (PCRs) provided by TPM to detect the forking attack [53]. For example, an enclave application sets PCR when booting, which enables another enclave instance of the same application program to detect the change of PCR value. Software-based systems [16, 33, 40] usually integrate the forking defense with the rollback defense together.

9 CONCLUSION

In this paper, we present Narrator, a system using the decentralized trust to provide performant state continuity protection for cloud TEEs. Narrator relies on a blockchain to initialize a distributed system of cloud TEEs, which adopts a customized consistent broadcast protocol to provide fast and unlimited state update and read. Our evaluation suggests that Narrator provides a practical solution for preserving state continuity of TEEs in the cloud.

REFERENCES

[1] AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/. Retrieved Jun, 2022.

[2] Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/. Retrieved Jun, 2022.

[3] Bitcoin Average Transaction Fee. https://ycharts.com/indicators/bitcoin_average_transaction_fee. Retrieved Jun, 2022.

[4] Google Could confidential computing. https://cloud.google.com/confidential-computing. Retrieved Jun, 2022.

[5] Intel SGX DCAP Orientation Guide. https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/DCAP_ECDSA_Orientation.pdf. Retrieved Jun, 2022.

[6] Latency in the Data Center Matters. https://www.networkworld.com/article/2230157/latency-in-the-data-center-matters.html. Retrieved Jun, 2022.

[7] SGX documentation: SGX create monotonic counter. https://software.intel.com/en-us/node/696638. Retrieved Jun, 2022.

[8] Technology Preview for secure value recovery. https://signal.org/blog/secure-value-recovery/. Retrieved Jun, 2022.

[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, pages 30:1–30:15, 2018.

[10] Architecure ARM. Security technology building a secure system using TrustZone technology (white paper). *ARM Limited*, 2009.

[11] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. Avocado: A secure in-memory distributed storage system. In *2021 USENIX Annual Technical Conference (USENIX ATC)*, pages 65–79, 2021.

[12] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 267–283, 2015.

[13] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 53–64, 2010.

[14] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[15] 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)*, pages 157–168, 2017.

[16] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Trusted computing meets blockchain: Rollback attacks and a solution for Hyperledger Fabric. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 324–32409, 2019.

[17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[18] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[19] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2nd edition, 2011.

[20] Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, 1999.

[21] Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi. Solidus: Confidential distributed ledger transactions via pvorm. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 701–717, 2017.

[22] Guoxing Chen and Yinqian Zhang. MAGE: Mutual attestation for a group of enclaves without trusted third parties. In *31st USENIX Security Symposium (USENIX Security)*, 2022.

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

[24] Phil Daian, Rafael Pass, and Elaine Shi. Snow White: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Financial Cryptography and Data Security (FC)*, pages 23–41, 2019.

[25] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Advances in Cryptology – EUROCRYPT*, pages 66–98, 2018.

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, pages 205–220, 2007.

[27] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. Damysus: Streamlined bft consensus leveraging trusted components. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, page 1–16, 2022.

[28] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 51–68, 2017.

[29] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security)*, pages 217–233, 2017.

[30] Stephen Hemminger et al. Network emulation with NetEm. In *Linux conf au*, page 2005, 2005.

[31] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[32] Mohit Kumar Jangid, Guoxing Chen, Yinqian Zhang, and Zhiqiang Lin. Towards formal verification of state continuity for enclave programs. In *30th USENIX Security Symposium (USENIX Security)*, pages 573–590, 2021.

[33] Gabriel Kaptchuk, Ian Miers, and Matthew Green. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[34] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-end integrity protection for web applications. In *2016 IEEE Symposium on Security and Privacy (S&P)*, pages 895–913, 2016.

[35] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, pages 51–58, 2001.

[36] Derek Leung, Adam Suhl, Yossi Gilad, and Nickolai Zeldovich. Vault: Fast bootstrapping for the algorand cryptocurrency. In *Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[37] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: A secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, page 63–79, 2019.

[38] Jian Liu, Wenting Li, Ghassan O Karame, and N Asokan. Scalable Byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68:139–151, 2018.

[39] André Martin, Cong Lian, Franz Gregor, Robert Krahn, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. ADAM-CS: Advanced asynchronous monotonic counter service. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 426–437, 2021.

[40] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security)*, pages 1289–1306, 2017.

[41] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Working Paper*, 2008.

[42] Neha Narula, Willy Vasquez, and Madars Virza. zkLedger: Privacy-Preserving auditing for distributed ledgers. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 65–80, 2018.

[43] Jianyu Niu, Wei Peng, Xiaokuan Zhang, and Yinqian Zhang. Narrator: Secure and practical state continuity for trusted execution in the could. https://github.com/teecertlab/papers/tree/main/Narrator, 2022. Retrieved Jun, 2022.

[44] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 Usenix Annual Technical Conference (ATC)*, pages 227–240, 2018.

[45] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. *SIGARCH Comput. Archit. News*, 37(1):97–108, 2009.

[46] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC)*, pages 305–319, 2014.

[47] Alina Oprea and Michael K Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *16th USENIX Security Symposium (USENIX Security)*, pages 183–198, 2007.

[48] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *2011 IEEE Symposium on Security and Privacy (S&P)*, pages 379–394, 2011.

[49] Michael K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS)*, page 68–80, 1994.

[50] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limited objects using a tpm without a trusted os. In *Proceedings of the First ACM Workshop on*

Scalable Trusted Computing (STC), page 27–42, 2006.

[51] AMD SEV-SNP. Strengthening VM isolation with integrity protection and more. *White Paper, January*, 2020.

[52] Raoul Strackx, Bart Jacobs, and Frank Piessens. ICE: A passive, high-speed, state-continuity scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, page 106–115, 2014.

[53] Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *25th USENIX Security Symposium (USENIX Security)*, pages 875–892, 2016.

[54] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security)*, pages 991–1008, 2018.

[55] Marten van Dijk, Jonathan Rhodes, Luis F. G. Sarmenta, and Srinivas Devadas. Offline untrusted storage with immediate detection of forking and replay attacks. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing (STC)*, page 41–48, 2007.

[56] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger Byzantium version. *Ethereum project yellow paper*, pages 1–32, 2018.

[57] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 347–356, 2019.

[58] Jiabin Zhu, Xiong Yan, and Wenchao Huang. A formal framework for state continuity of protected modules. In *2018 4th International Conference on Big Data Computing and Communications (BIGCOM)*, pages 114–119, 2018.