ENGRAFT: Enclave-guarded Raft on Byzantine Faulty Nodes

(Full Version)

Weili Wang*†
12032870@mail.sustech.edu.cn
Southern University of Science and Technology

Sen Deng*†
12032873@mail.sustech.edu.cn
Southern University of Science and Technology

Jianyu Niu†
niujy@sustech.edu.cn
Southern University of Science and Technology

Michael K. Reiter
michael.reiter@duke.edu
Duke University

Yinqian Zhang†‡
yinqianz@acm.org
Southern University of Science and Technology

ABSTRACT

This paper presents the first critical analysis of building highly secure, performant, and confidential Byzantine fault-tolerant (BFT) consensus by integrating off-the-shelf crash fault-tolerant (CFT) protocols with trusted execution environments (TEEs). TEEs, like Intel SGX, are CPU extensions that offer applications a secure execution environment with strong integrity and confidentiality guarantees, by leveraging techniques like hardware-assisted isolation, memory encryption, and remote attestation. It has been speculated that when implementing a CFT protocol inside Intel SGX, one would achieve security properties similar to BFT. However, we show in this work that simply combining CFT with SGX does not directly yield a secure BFT protocol, given the wide range of attack vectors on SGX. We systematically study the fallacies in such a strawman design by performing model checking, and propose solutions to enforce safety and liveness. We also present ENGRAFT, a secure enclave-guarded Raft implementation that, firstly, achieves consensus on a cluster of $2f + 1$ machines tolerating up to f nodes exhibiting Byzantine-fault behavior (but well-behaved enclaves); secondly, offers a new abstraction of confidential consensus for privacy-preserving state machine replication; and finally, allows the reuse of a production-quality Raft implementation, BRaft, in the development of a highly performant BFT system.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; **Formal security models**.

KEYWORDS

fault tolerance; model checking; trsuted execution environments

*Equal contribution.
†Affiliated with the Research Institute of Trustworthy Autonomous Systems and the Department of Computer Science and Engineering.
‡Corresponding author.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560639

ACM Reference Format:
Weili Wang, Sen Deng, Jianyu Niu, Michael K. Reiter, and Yinqian Zhang. 2022. ENGRAFT: Enclave-guarded Raft on Byzantine Faulty Nodes: (Full Version). 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, 17 pages. https://doi.org/10.1145/3548606.3560639

1 INTRODUCTION

Consensus algorithms have been employed in modern production systems to increase their reliability and availability, by replicating data and computation across a group of computers. Sophisticated protocols must be implemented to ensure the group collectively behaves as if it is a single machine, internally reaching agreement on the computation state. Consensus algorithms are also seen in decentralized computing systems such as permissioned blockchains, which employ a group of machines to distribute trust among multiple mutually distrusting entities.

A classic abstraction of such distributed systems is state machine replication [55], where data and computation are replicated among multiple machines and transitions among states are coordinated such that non-faulty machines maintain consistent copies of the state machine. Crash fault-tolerant (CFT) coordination protocols ensure such consistency despite failures including machine crashes, network faults, and network partitions, while Byzantine fault-tolerant (BFT) protocols additionally overcome a limited number of malicious replica compromises, operator mistakes, and software errors. As such, BFT protocols usually have more complicated communication patterns and worse performance. BFT protocols (*e.g.*, PBFT [20]) can tolerate up to f faulty machines among $3f + 1$ machines, whereas most CFT protocols (*e.g.*, Paxos [38] and Raft [48]) tolerate up to f faulty machines out of $2f + 1$ machines.

The emergence of commercial hardware support for trusted execution environments (TEEs) or "enclaves" (*e.g.*, SGX), which isolate software and data from compromises and mistakes on the host platforms, suggests an alternative design point beyond these CFT and BFT options. In brief, by executing a CFT protocol within enclaves, one might achieve a combination of properties: assured safety (consistency) despite Byzantine faults in the platforms hosting the enclaves—but not *within* the enclaves—and assured liveness (progress) and performance provided that at least a majority of

enclaves remain functional and are permitted unfettered communication with one another. Indeed, this alternative has been leveraged in a number of efforts (e.g., [11, 15, 29, 54]), implicitly assuming that this combination provides the safety, liveness, and performance properties mentioned above. For instance, Signal integrates Raft into SGX for secure value recovery [11], TEEKAP [29] leverages in-enclave Raft and threshold secret sharing to build self-expiring data objects, and CCF [54] ports Raft into SGX for permissioned confidential blockchains.

In this paper, we question this basic premise and perform the first critical evaluation of the conditions under which CFT and TEEs together provide the promises outlined above. By leveraging automated model-checking tools, TLC [69], we identify the assumptions made in a popular and widely used CFT design, Raft [48], to achieve safety and liveness on crash-faulty machines and reveal incongruities between these assumptions and the assurances that SGX provides on Byzantine faulty nodes. First, SGX's lack of state continuity opens the door to rollback attacks, which are fatal for CFT protocols depending on persistent storage (*e.g.*, Raft) since they assume persistent data is always fresh, which is true in CFT. Second, the timeout-driven leader election mechanism in Raft can be manipulated by an adversary with system privileges, which leads to compromises of livenss.

We hence propose mitigations to close these gaps, thereby placing this conventional wisdom about "CFT+TEEs" on a more solid footing. For instance, to preserve state continuity of Raft nodes and prevent rollback attacks, we propose TIKS, a distributed in-memory key-value storage inside SGX enclaves for storing Raft meta data. Compared to alternative solutions, such as hardware monotonic counters [13, 59] and ROTE [45], TIKS is the only practical solution in cloud settings that offers both state continuity and recoverability.

Informed by this experience, we then report our design and implementation of ENGRAFT, a secure ENclave-Guarded Raft implementation that reuses a production-quality Raft implementation, BRaft [1], in the development of a highly performant BFT system. ENGRAFT achieves consensus on a cluster of $2f + 1$ machines tolerating up to f nodes exhibiting Byzantine-fault behavior.

We provide a prototype implementation of ENGRAFT with 3kLoC C++ code atop the BRaft code base. We evaluate ENGRAFT with Intel SGX in both LAN and WAN settings using virtual machines in three geo-distributed data centers hosted in a public cloud. We compare the performance of ENGRAFT with both BRaft and `ChainedDamysus` [25], a recent BFT system that leverages SGX to improve resilience (but not confidentiality). The results show that ENGRAFT achieves comparable performance. The source code for ENGRAFT is released in [4].

To summarize, our contributions are as follows:

- We present a systematic analysis of the safety and liveness issues behind the idea of combining CFT protocols and TEEs to achieve highly performant and confidential BFT consensus.
- We leverage model checking to assist automated identification of safety and liveness violations caused by threats specific to SGX. Our work provides the first example of the modeling and checking of these properties under such a threat model.
- We propose TIKS, a distributed in-memory key-value storage for rollback prevention. Compared to alternative solutions, it offers both state continuity and recoverability.
- We implement and evaluate ENGRAFT, a system that reuses a production-quality Raft implementation to achieve consensus on a cluster of $2f + 1$ nodes tolerating up to f Byzantine faulty nodes.

2 BACKGROUND

2.1 Consensus Algorithms

2.1.1 Crash fault tolerance (CFT). A machine suffers a crash fault if it halts execution prematurely. The seminal work by Lamport [38, 39], referred to as Paxos, has served as a general solution to crash fault tolerance. However, Paxos is difficult to understand and to implement correctly in practice. To overcome this, Ongaro and Ousterhout proposed Raft [48], a CFT algorithm that is easy to understand and has been implemented in many practical systems such as distributed databases [14].

As a representative CFT protocol, Raft consensus can support a cluster of $2f + 1$ machines with at most f faulty machines. In Raft, a node may stay in one of the three states, *follower*, *candidate* and *leader*. Raft nodes uses two basic RPCs, *i.e.*, `AppendEntries` and `RequestVote` to maintain the consistency of a persistent log storage, and every log entry records its *term*, its *index*, and a command to be applied on the state machine. The term and log index serve as logical clocks in the Raft protocol.

Raft log replication. In a normal case, there is one leader in the cluster interacting with clients and multiple followers replicating the log. Upon receiving a client request, the leader generates an uncommitted log, replicates it by sending an `AppendEntries` RPC to the followers, marks it as committed after receiving responses from the majority of the followers, applies it to the state machine, and finally replies to the client.

Raft leader election. To recover from leader crashes, followers set up election timers and become candidates once timers expire. As such, the leader needs to send heartbeat signals (by issuing `AppendEntries` RPCs with empty log entries) to followers periodically to legitimate its leadership (by resetting followers' election timer). In the leader election, a candidate firstly increases its term, votes for itself, and then requests votes from other followers. A follower will vote for a candidate if and only if the following conditions hold: (1) The follower has not voted for any other candidates of the same term. (2) The candidate's term and index of the last log must be as new as the follower.

The above constraints ensure that at any term, only one candidate can receive votes from the majority and become the leader. If a candidate fails to collect votes, it will repeat the above procedure (increase term and request votes) indefinitely until it succeeds, discovers the leader at the current term, or observes a higher term.

To recover from the crash, Raft persists internal states and then recovers from them. Specifically, the current term of a state machine, vote information (whom it voted for), and replicated logs are stored on the disk.

2.1.2 Byzantine fault tolerance (BFT). Lamport *et al.* [41] first proposed the Byzantine generals problem, in which a group of machines has to reach agreement of the same output in the presence of malicious attacks, operator mistakes, and software errors. Compared with CFT protocols, BFT protocols usually have more complicated communication mechanisms and worse performance because of dealing with arbitrary faults. BFT protocols can tolerate up to f faulty machines among $3f+1$ machines (instead of $2f+1$ machines in CFT protocols).

2.2 Software Guard Extension

Intel Software Guard eXtensions (SGX) [30] is the most prevalent TEE realization, intending to provide shielded execution environments, *i.e.*, enclaves, for programs. In summary, SGX offers the following security guarantees.

Isolation. SGX programs can be divided into trusted and untrusted components. ECALL interfaces are called from the host to enter the enclave, while OCALL interfaces are called inside the enclave to request untrusted services like system calls.

Sealing. SGX supports sealing, a procedure using the sealing key to encrypt in-enclave data and storing it in the external storage. With an optional strict key derivation policy, the sealing key is only accessible to the enclave creating it.

Attestation. SGX provides both local and remote attestation to facilitate the identification of enclaves. The local attestation is conducted to verify whether the counterpart is running inside the enclave on the same platform, while the remote attestation is used to tell whether the remote party is shielded by legitimate enclaves with expected properties (*e.g.*, updated microcode and specified enclave identity).

2.3 Model Checking

Model checking [46] is a technique that can model a system and determine whether the system satisfies given security properties. The input of the model checker is usually a specification of the system and some expected properties. The model checker can verify whether the properties are met or not. If not, it reports a specific counterexample: an execution-trace that violates the property.

TLA+. TLA+ is a high-level formal specification language developed by Lamport [40] for modeling and validating programs and systems, especially for concurrent and distributed ones based on TLA (Temporal Logic of Actions) [37]. TLA+ provides a uniform mathematical language to model the systems or properties.

- *Variables and Constants*: Users can specify multiple variables when modeling system states, which will change as the state changes. By contrast, constants remain the same in all states.
- *States, Steps, Behaviors*: The set of specified variables constitutes the state. A pair of successive states form a step, and a continuous sequence of states forms a behavior.
- *Specification*: The specification of a system describes the state transition over time. A canonical form of a TLA+ specification is:

$$Spec \triangleq Init \wedge \Box[Next]_{vars},$$

where *init* is the initial state, *vars* is a tuple of variables, and *next* is an action. *Spec* denotes the entire state space of the system that one wants to check. Starting from the initial state, one action at a time is elected to transfer the system state to another until the entire state space is traversed.

TLC. TLC is an explicit-state model checker implemented by Yu, *et al.* [69] to check both safety and liveness properties of TLA+ specifications. TLC can perform two basic modes: simulation and model checking. The simulation mode begins with a random initial state and then repeatedly chooses a next state to a depth specified by the user, while the model-checking mode tries to check all possible behaviors and build the graph of all reachable states using breadth-first search.

3 OVERVIEW

3.1 Problem Statement

In this paper, we investigate the security properties of a CFT protocol (*i.e.*, Raft) running inside trusted execution environments (*i.e.*, Intel SGX). Intuitively, by combining the integrity guarantees provided by SGX and the crash tolerance provided by CFT, one would easily achieve consensus even with Byzantine behaviors. For instance, Avocado [15] explicitly claims that "*We design a secure replication protocol, which builds on top of any high-performance non-Byzantine protocol–our key insight is to leverage TEEs to preserve the integrity of protocol execution, which allows to model Byzantine behavior as a normal crash fault.*" Similar assumptions have been made in some other studies [11, 29, 54].

However, this seemingly correct statement might not withstand scrutiny, since it has been shown that SGX is vulnerable to various attacks (including state replay attacks [45]). Therefore, it is expected that a CFT implementation inside SGX enclaves is also susceptible to these threats; yet it is unclear to what extent is such a strawman design vulnerable and, if so, how to fix these issues. Therefore, the research goals of this paper are twofold.

Security analysis: Given the threat model of Intel SGX, we will investigate whether a vanilla Raft implementation inside SGX enclaves could achieve safety and liveness properties. The answer will bring forth the following new insights into the development of TEE-assisted distributed systems.

- We explore the security impacts of SGX-related threats on distributed protocols. To the best of our knowledge, such investigation has never been performed in prior studies.
- We leverage formal methods (e.g., model checking) to assist the automated identification of safety and liveness issues caused by these SGX-related threats. It is yet unclear how such threats can be modeled and checked with model checking tools.
- We design new solutions to mitigate SGX-related threats in the distributed settings. While some countermeasures have been proposed in the literature, our aim is to minimally patch the design of Raft with countermeasures customized for Byzantine settings.

System design: The paper also aims to build a highly secure, performant, and confidential BFT protocol (which we call ENGRAFT) by porting a production-quality CFT implementation into SGX enclaves. We envision three important merits of the resulting protocol:

- *Highly secure.* Whereas most BFT protocols assume less than 1/3 faulty nodes to function, CFT protocols relax the requirement to 1/2, yielding a more practical and scalable solution for real-world use cases. A BFT solution that combines CFT and SGX would achieve the same effect with Byzantine fault assumptions.
- *Performant.* CFT protocols, such as Raft, have been thoroughly studied and understood; production-quality implementations of CFT protocols, such as BRaft [1], have been well maintained as open-source projects and used widely in real-world settings. Moreover, SGX is a hardware feature commercially available on most server-end Intel processors[1]. A straightforward integration of a highly performant BRaft with Intel SGX would satisfy most industry-level use cases.
- *Confidential.* SGX naturally provides confidentiality to code and data inside enclaves. Therefore, by porting the CFT protocol into enclaves, confidentiality of the state replicas can be achieved as a by-product.

3.2 Threat Model

We consider a distributed system with a set of $n = 2f + 1$ SGX-enabled machines that collectively provides services to clients in partially synchronous networks. We assume at most f machines can be controlled by a malicious operator $\mathcal{A}$ at any time. The malicious operator $\mathcal{A}$ can launch, suspend, resume, and terminate SGX enclaves at her will. She also controls the CPU scheduling, memory management, and I/O operations. She can conduct attacks, for example, to breach the enclaves' state continuity [31], by providing the enclaves with stale persistent states (*e.g.*, older versions of encrypted files) instead of the latest ones.

However, we assume SGX is secure and software protected within SGX enclaves preserves its integrity and confidentiality during its computation, by leveraging techniques such as memory isolation, encryption and remote attestation. Specifically, transient execution attacks [22, 57, 61] that leak enclave memory and attestation keys are outside the scope of this paper, as Intel has already provided microcode patches that mitigate these vulnerabilities [8]. Micro-architecture side-channel attacks that leak secret values by observing enclave memory access patterns are still possible. But we assume countermeasures have been implemented at the software level, especially for cryptographic libraries (as is the case of OpenSSL and Intel SGX SSL). As such, the corrupted machines can only affect the availability of the software inside the enclaves but not its correctness.

4 SECURITY ANALYSIS

In this section, we perform a systematic analysis of the core component of Engraft—SgxBraft, a vanilla porting of BRaft inside enclaves—in the context of SGX-related security threats. Our analysis is assisted by a model checking tool, which models a comprehensive list of attack vectors and automates the analysis to identify vulnerabilities.

[1]While Intel has discontinued support of SGX in 12th Core processors, SGX will be continued in server-end processors [6].

Table 1: Byzantine behaviors via filesystem manipulation.

Items	Explanation
fs_currentTerm-	decrease variable *currentTerm*
fs_currentTerm+	increase variable *currentTerm*
fs_votedFor-	drop vote information
fs_votedFor+	vote for other servers
fs_log-	drop *log entries*
fs_log+	append *log entries*

Table 2: Byzantine behaviors via network manipulation.

Items	Explanation
nw_RequestVote_term-	decrease *term*
nw_RequestVote_term+	increase *term*
nw_RequestVote_lastLog-	decrease *lastLog*
nw_RequestVote_lastLog+	increase *lastLog*
nw_AppendEntries_term-	decrease *term*
nw_AppendEntries_term+	increase *term*
nw_AppendEntries_preLog-	decrease *preLog*
nw_AppendEntries_preLog+	increase *preLog*
nw_AppendEntries_leaderCommit-	decrease *leaderCommit*
nw_AppendEntries_leaderCommit+	increase *leaderCommit*
nw_AppendEntries_entries+	modify *log entries*

4.1 Threat Modeling

The core logic of a Raft node is implemented inside an enclave, which protects its confidentiality and integrity. But it still needs to interact with the outside world to (1) save its state to the local disk for crash recovery, and to (2) communicate with other nodes to maintain the consensus of the distrusted system. These interactions introduce additional attack vectors to SgxBraft. We categorize these attack vectors into two classes: filesystem manipulation and network manipulation.

- *Filesystem manipulation.* In the Raft protocol, three variables are persistent in the filesystem: *currentTerm*, *votedFor* and *log*. Upon crash, Raft needs to restore these persistent variables from the disk to recover its state. The adversary that controls the OS could modify the persistent storage or provide a stale state to SgxBraft. We summarize the Byzantine fault behaviors via filesystem manipulation in Table 1.
- *Network manipulation.* There are two types of RPC messages in Raft: `RequestVote` RPC and `AppendEntries` RPC. Without proper protection, the adversary is able to insert or modify the content of messages sent between the enclaves. We summarize Byzantine behaviors via network manipulation in Table 2.

4.2 Model Checking Safety Properties

To automate the analysis of the safety properties of SgxBraft, we leverage the TLC model checker [69]. Similar to other formal methods, model checking requires explicit specification of attack primitives that the adversary relies upon to construct sophisticated attack steps with which the desired properties are eventually

compromised. To systematically enumerate all attack primitives considered in our threat model, we classify all possible Byzantine behaviors into two categories (see Sec. 4.1) and carefully examine how each of the key elements in the Raft protocol can be manipulated by Byzantine behaviors of either category. Accordingly, we extend the TLA+ formal specification of Raft from the original Raft paper [48] with 17 Byzantine behaviors listed in Table 1 and Table 2. Each of these Byzantine behaviors is modeled as an action in TLA+. For each Byzantine behavior, we use the TLC model checker to examine four safety properties of Raft [48]: election safety, log matching, leader completeness, and state machine safety.

We derive the TLA+ specification of the four safety properties and list them in Appendix A. It is worth noting that while the original work by Ongaro has provided the TLA+ specifications of the Raft protocol, it did not perform checking using the TLC model checker (see [47, sec. 8.2]). Therefore, our work provides the first successful attempt to formally check the four safety properties of Raft using TLC.

We note that we employ model checking primarily as a tool for automated extraction of attack traces if there are any. TLC cannot be used to prove the correctness of SGXBRAFT. This is because the unbound variables (*e.g.*, terms, log entries) used in Raft lead to an infinite search space, which cannot be exhausted by TLC. In fact, while model checking has been used to validate specific properties of simple network protocols [27], it is rarely used to prove correctness of complex distributed systems. The most common use of model checkers is to detect bugs in such designs [18, 36].

TLC setup. In its configuration, TLC requires specifying the number of total nodes and malicious nodes in the cluster. Without loss of generality, in each test case, we model and check two configurations: a three-node cluster with one malicious node (i.e., $f = 1$) and a five-node cluster with two malicious nodes (i.e., $f = 2$). All tests were run in the TLC simulation mode. When the search space exceeded twenty billion states for each configuration, the search process was terminated. The result of model checking was a list of attack traces that would lead to violation of one of the safety properties. If no attack traces are reported, it suggests TLC fails to identify attacks within the twenty billion states. On average, each test finished in approximately eight hours on a platform equipped with 32GB memory and an Intel Core i7-10700 CPU.

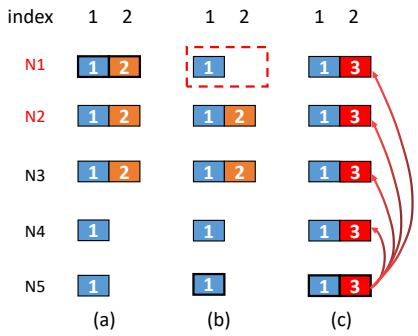


Figure 1: Illustration of an attack traces for Byzantine behavior #5: fs_log-.

Model checking results. TLC reported attack traces for 8 out of the 17 Byzantine behaviors we specified. The results are shown in Table 3. Five Byzantine behaviors caused by filesystem manipulation can violate one of the safety properties, while TLC reported three of eleven Byzantine behaviors related to network manipulation are effective.

Case study. We illustrate the attacks due to Byzantine behavior #5 in Fig. 1. In Byzantine behavior #5, the adversary provides stale log entries when recovering from crashes. As shown in the figure, N1 and N2 are malicious nodes; N3, N4, and N5 are benign. In Fig. 1(a), all nodes are at term 2, which is recorded in the persistent storage. N1 is the leader that replicates the log entry to N2 and N3. At this point, the log entry of term 2 has been replicated to a majority of nodes, and therefore it is considered committed.

Next, as shown in Fig. 1(b), N1 is crashed by the adversary and restarted with stale log entries (per Byzantine behavior #5). The consequences of the attack are twofold: (1) The log entry of index 2 is lost on N1; and (2) N1 becomes a follower. As such, a leader election is triggered. As we assume in this case the adversary only rolls back log entries but not the terms, all nodes advances to term 3. Although N2 and N3 have newer logs and hence will not vote for N5, it is still possible for N1, N4, and N5 to vote for N5. Assume N5 is elected as the new leader for term 3.

In Fig. 1(c), N5 receives the client request and appends the log with term 3. It then replicates the log to all other nodes, causing the log entry from term 2 to be overwritten. Two out of four safety properties were violated: leader completeness, as the new leader N5 doesn't have the log entry committed from term 2, and state machine safety, as N2 and N5 apply different log entries at index 2.

Countermeasures. To counter the safety violations due to filesystem and network manipulation listed in Table 3, we propose three categories of countermeasures.

- *File encryption.* The root cause of Byzantine behavior #4 and #6 is that the adversary can arbitrarily modify the content of persistent files, which can be prevented using file encryption (see Sec. 6.2.1).
- *Rollback prevention.* Byzantine behavior #1, #3 and #5 can be launched by rollback attacks, which can be prevented using state continuity mechanisms (see Sec. 5.1).
- *Network encryption and authentication.* Byzantine behavior #10, #13, #15 can be prevented using authentication and encrypted channels between nodes (see Sec. 6.2.2).

4.3 Model Checking Liveness Properties

We perform model checking with TLC to systematically analyze the liveness properties of SGXBRAFT. To our knowledge, this is the first attempt to model check liveness properties of consensus protocols. We specifically consider the liveness property in the leader election phase. We build models for two separate cases, one with the *preVote* mechanism [47] and one without. As Raft's leader election procedure relies on timeout mechanisms extensively, we model one timer for each node using a TLA+ variable. We model the passage of time as a TLA+ action, which decreases the timer value by one for all non-leader nodes. Once a timer expires, *i.e.*, decreases to zero, it sends a `RequestVote` RPC (modelled as a TLA+ action) to all other nodes.

Table 3: Analysis results for different Byzantine behaviors.

No.	Byzantine Behaviors	Safety Properties			
		Election Safety	*Log Matching*	*Leader Completeness*	*State Machine Safety*
1	fs_currentTerm-	×	×	×	×
2	fs_currentTerm+	-	-	-	-
3	fs_votedFor-	×	×	×	×
4	fs_votedFor+	×	×	×	×
5	fs_log-	-	-	×	×
6	fs_log+	-	×	×	×
7	nw_RequestVote_term-	-	-	-	-
8	nw_RequestVote_term+	-	-	-	-
9	nw_RequestVote_lastLog-	-	-	-	-
10	nw_RequestVote_lastLog+	-	-	×	×
11	nw_AppendEntries_term-	-	-	-	-
12	nw_AppendEntries_term+	-	-	-	-
13	nw_AppendEntries_preLog-	-	×	×	×
14	nw_AppendEntries_preLog+	-	-	-	-
15	nw_AppendEntries_entries	-	×	×	×
16	nw_AppendEntries_leaderCommit-	-	-	-	-
17	nw_AppendEntries_leaderCommit+	-	-	-	-

Legend:×: Safety Properties Violated. - : TLC can't report violation within twenty billion states, running ten times.

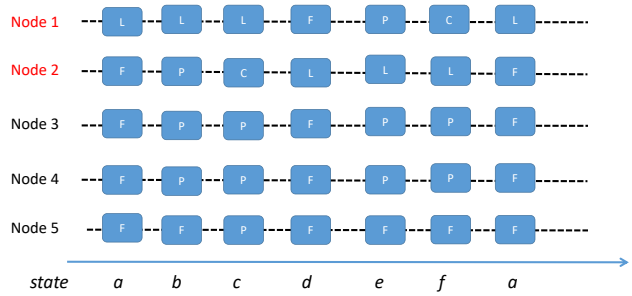


Figure 2: Illustration of an attack trace that breaks liveness in leader election. Node 1 and node 2 are malicious nodes while the others are benign. *L*, *F*, *P*, **and** *C* **denote** *Leader*, *Follower*, *preCandidate* **and** *Candidate* **respecitively.**

As with the case of safety checking, each model contains a five-node cluster with two malicious nodes. Our analysis only considers one attacker behavior—dropping messages to and from the enclave on the malicious nodes. Therefore, we model such behavior as an action in TLA+. The liveness property considered is that a benign node can be eventually elected as leader. This is modelled as a temporal property, as shown below:

$$<> \exists i \in Benign\_Server : state[i] = Leader$$

where *Benign_Server* represents the set of all benign nodes and $state[i] = Leader$ denotes that node i is the *Leader*; <> represents temporal logic "*Eventually*" in TLA+.

Model checking results. TLC reports a liveness property violation for the leader election procedures, both with and without *preVote*. The root cause in both cases is that the Raft leader election procedure relies on Heartbeat-based timeout and request-vote mechanisms, which are susceptible to manipulation by the OS.

The attack trace generated for leader election with *preVote* is listed in Fig. 2. Node 1 and node 2 are malicious while the others are benign. At state *a*, node 1 is the leader. It discards all Heartbeat messages sent to the node 2, 3, and 4; hence all followers except node 5 become the preCandidate at state *b*. The preCandidate cluster consisting of node 2, 3 and 4 votes for each other to become candidates. However, node 2 drops the vote requests from node 3 and node 4, such that node 2 does not vote for nodes 3 and 4. As a result, only node 2 can obtain the majority votes (*i.e.*, three votes) to become a candidate with the current term plus one at state *c*. Then it sends requestVote messages to all other nodes to collect votes, and is elected as the new leader at state *d*. After that, node 2 repeats the above steps, and node 1 becomes leader again, causing the entire system to return to its initial state *a*. As malicious node 1 and node 2 may take turns to become the leader, the liveness property is violated.

Countermeasures. To ensure that the benign node can be elected as leader, we introduce a client alert mechanism such that leader elections can be triggered by mechanisms other than the absence of heartbeats and that the benign nodes have a chance of winning the election (see Sec. 5.2).

5 SYSTEM DESIGN

In this section, we present our design of ENGRAFT that provides a highly secure, performant, and confidential BFT implementation. ENGRAFT consists of three components: SGXBRAFT, a vanilla implementation of BRaft inside enclaves; TIKS, a Trustworthy distributed In-memory Key-value Storage (KV store); and MLD, a mechanism that detects and preempts misbehaving leaders. The latter two components are proposed to counter the safety and liveness violations we discovered in Sec. 4, respectively.

5.1 TIKS: Rollback Prevention

To provide rollback prevention, a trustworthy monotonic counter service is necessary. However, monotonic counter APIs (available in Linux SGX SDKs before v2.9) exposed to SGX enclaves require Intel Converged Security and Management Engine (CSME) [21], an Intel chipset with controversial security concerns [5, 7, 28]. Recent Intel processors, especially server-end products, are not equipped with CSME. Moreover, monotonic counter APIs have been removed from SDKs since v2.9. Therefore, current Intel SGX platforms do not offer any trustworthy means for rollback prevention. Moreover, leveraging hardware TPMs drastically expands the TCB and introduces issues like *Cuckoo attack* [49]. As such, building a distributed subsystem for preventing state rollback appears to be the only viable solution. In this paper, we propose TIKS, a trustworthy in-memory KV store tightly integrated in ENGRAFT to prevent rollback attacks.

5.1.1 Persistent Files. According to Table 3, all persistent variables in Raft are vulnerable to rollback attacks, including *currentTerm*, *votedFor* and *log*. As such, in BRaft, the following two persistent files need to be guarded with freshness.

- *Raft meta file.* The Raft meta file contains two variables related to voting, *i.e.*, *votedFor* and *currentTerm*.
- *Log data file.* The log data file stores all log entries.

5.1.2 Rollback Protection Requirements. One would hope to ensure freshness of the Raft meta file and the log data file by using trusted monotonic counters. Unfortunately, trusted monotonic counters are not panaceas [59]. If the node increases the counter first and then persists the data together with the increased counter value, there is a short time window between the counter increment and the data persistency in which a node cannot recover from crashes [45]. This is because there is no persisted data corresponding to the updated counter. If one chooses to persist the data before increasing the counter, it is possible for an adversary to bind the same counter value with multiple persistent files, breaking safety [59].

This "*recoverability* or *safety*" dilemma is unavoidable when using trusted counters to preserve state continuity. The root cause is that counter increment and data persistence are two separate operations. To securely use monotonic counters for the desired purposes, one must ensure atomicity when combining the two operations. In ENGRAFT, we propose TIKS to substitute trusted counters and provide a *safe* and *recoverable* rollback protection.

5.1.3 TIKS Overview. The design goal of TIKS is not to build a generic solution for rollback prevention. Rather, the design is customized for ENGRAFT. Particularly, since there are at least $f + 1$ benign and operational nodes out of the total $2f + 1$ nodes in a ENGRAFT cluster, it is possible to store data in a distributed storage by leveraging the $f + 1$ correct nodes, instead of the local disks. TIKS adopts a customized version [45] of the echo broadcast protocol [53], which we will describe in Sec. 5.1.5.

The Raft meta file can be stored in TIKS directly, since *votedFor* and *currentTerm* are small variables that take less than a hundred of bytes. However, the size of the log data file increases rapidly when appending logs, which is too big to be stored in TIKS. Therefore, we create a new file, the log meta file, to store the hash value of the log data file, and store the log meta file in TIKS while keeping the log data file on the local disks. As updates of the log data file are frequent, calculating its hash value frequently would become inefficient. Therefore, we adopt a *chained hash* design for the log meta file. We will describe this in detail in Sec. 5.1.7.

5.1.4 Distributed Key-value Storage. TIKS is a distributed in-memory KV store that protects its data inside the enclaves. A node of ENGRAFT also serves as a node of TIKS. TIKS only stores two files, the Raft meta file and the log meta file. Therefore, in a three-node cluster, a total of six files are stored in TIKS, which are associated with six different keys. Moreover, we attach a monotonic index to each file to indicate its latest version and thus a key-value item in the KV store has a structure of $\langle key, \langle index, file\rangle\rangle$, where $\langle index, file\rangle$ is a 2-tuple and *file* stores either a Raft meta file or a log meta file. We design two schemes, *storage update* and *recovery*, on top of this KV store abstraction, to ensure that a crashed node can recover its state. We detail these two schemes next but defer their security analysis to Appendix B.1.

5.1.5 Storage update. To enforce immutability, which requires that any KV store updates will be reflected in the subsequent reads, TIKS adopts a two-round communication protocol. Specifically, after the storage updating node updates its own KV store, it performs two RPCs to update other KV stores: `Store` and `ConfirmStore`. The workflow is shown in Fig. 3, where Node 1 in a three-node cluster is updating its state.

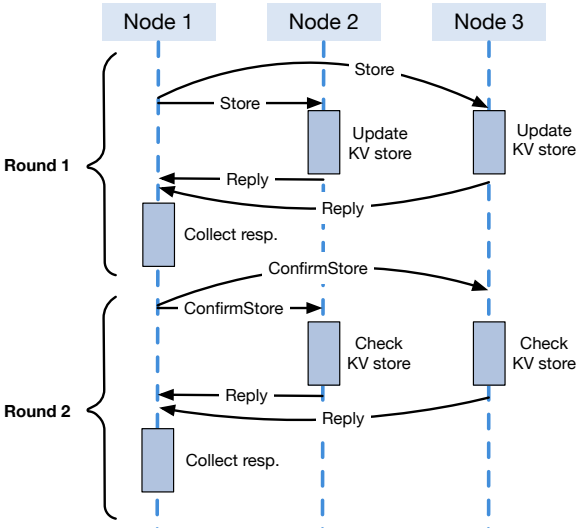


Figure 3: Workflow of storage update in TIKS.

- *The first round.* First, the updating node specifies the key of the to-be-updated file (either a Raft or log meta file) and its content in the `Store` RPC requests. A correct node receiving the `Store` RPC request will faithfully update its own KV store and return a response. If the updating node can collect at least f responses, it then passes the first round. Otherwise, it re-sends the `Store` RPCs.
- *The second round.* To prevent a rollback attack against TIKS, a second-round communication is necessary. Specifically, the updating node sends a `ConfirmStore` RPC request to f nodes that have responded in the first round, with the same content as the `Store`

RPC. A correct node that receives a `ConfirmStore` request will reply with a boolean value to indicate success, if it has responded the `Store` RPC and its KV store contains an item identical to the one in the request; otherwise, it ignores the request. If the updating node finally collects at least f responses to its `ConfirmStore`, it is assured that the updated file has been safely stored in TIKS; otherwise, the write has failed and must be reattempted.

To update the log data file, which is stored on the disk, the updating node should perform a `fsync` system call before updating the log meta file in TIKS to maintain recoverability.

5.1.6 Storage Recovery. ENGRAFT nodes maintain the KV store by themselves in normal phase but they need to retrieve the distributed storage from others after crashes since the storage only resides in memory. The storage retrieval algorithm goes as follows.

When a node is not crashed, it could directly read from its own KV store to fetch data, since all updates to its own files must be in its own KV store (not necessarily true for others' files). But a crashed node has to follow the following steps for storage recovery:

- Step 1: *Inquiry*. The crashed node issues `RetrieveStorage` RPCs to all nodes in the cluster until it has successfully collected at least $f + 1$ responses. Upon receiving the `RetrieveStorage` request, a correct node should reply with its entire KV store.
- Step 2: *Reconstruction:* With at least $f + 1$ copies of the KV store, the requesting node reconstructs its own KV store one item at a time. For each item, it compares the index of the item from each received copy and picks one with the largest index. After reconstructing the KV store, the requesting node should write back its own states using `Store` and `ConfirmStore` RPCs.

When there are one or more aborted store attempts, index conflict may happen. For example, a node that has successfully stored its Raft meta file at index $l - 1$ may crash during the process of storing a newer Raft meta file (denoted as $RaftMeta_1$) at index l. After recovery, the node may restore its Raft meta file at index $l - 1$ and later crash again when storing a different Raft meta file (denoted as $RaftMeta_2$) at index l. In this case, during recovery, the node may observe $RaftMeta_1$ and $RaftMeta_2$ at the same index l. We call this situation *index conflict*.

By design, TIKS handles index conflicts by consulting with the ENGRAFT leader. Though the existence of index conflict impedes the selection of one file with the largest index, a recovering node is able to restore its term (*i.e.*, *currentTerm*).

First, *currentTerm* is set to the highest term appearing in the retrieved Raft meta files that have the largest index (denoted as l). Then, the recovering node keeps waiting for a leader whose term (denoted as $term''$) is equal or larger than its own, *i.e.*, $term'' \geq currentTerm$.

After discovering a leader, the recovering node handles index conflict as follows. First, if there is index conflict on the Raft meta file at index l, the node synchronizes its *currentTerm* to that of the leader, sets its *votedFor* as the leader, and then updates its Raft meta file at index l. Second, if there is an index conflict on the log meta file at the largest index k, the node synchronizes its log storage with the leader, and updates its log meta file at index k in TIKS.

5.1.7 Log Hash Summary. The log meta file stores the *chained hash* of the log data file. The final hash value of the chain is called the *log hash summary*, denoted as h. At any time, the log data file contains log entries from index a to index b, denoted $[log_a, log_b]$, where log_i is the log entry with index i. h_i denotes the chained hash value calculated up to log_i (hence $h = h_b$). h_0 is a bitstring of 256 zeros. The most common operations on the log is to append a log entry. When log_i is appended, the nodes compute $h_i = H(h_{i-1} + H(log_i))$, where H is the hash function and + denotes the concatenation of two bitstrings. Each h_i is stored temporarily for the last committed log entry log_i and any uncommitted log entries appended after it.

There are two scenarios in which log entries are removed [47]: (1) when a snapshot of the logs is taken, all committed log entries (the ones from the beginning) are deleted. (2) When uncommitted log entries conflict with the leader's, the conflicted log entries (the ones from the end) are removed. As such, to delete the log entries before log_a, the node updates $h_a = H(h_0 + H(log_a))$, and then updates h by re-computing the chained hash values of $[log_a, log_b]$. To delete log entries after log_b, the node simply makes $h = h_b$.

Log recovery. To recover from crash, a node first retrieves the log meta file from the KV store and obtains the log hash summary h. It then computes the last chained hash value from the log data file and compares it with h. Inconsistency indicates rollback attacks.

5.2 MLD: Malicious Leader Detector

MLD is designed to guarantee liveness in ENGRAFT. Since Raft adopts a strong leader model, a malicious leader is able to launch DoS attacks or censorship attacks. Moreover, by corrupting multiple nodes, the attacker can ensure that one of the malicious nodes she controls is elected as the leader at any time.

5.2.1 Liveness Requirements. Liveness mandates that a client's request is eventually executed by the server cluster. In ENGRAFT, liveness involves three requirements. Once the network becomes synchronous (eventually happens in the partial synchrony model) and these three requirements are met, liveness can be achieved [20, sec. 4.5.2].

1. *Misbehavior exposure:* There is a mechanism in place to preempt the current leader if its misbehavior, *i.e.*, not processing client requests in time, is evident.
2. *Election of a benign leader:* A benign leader is eventually elected.
3. *Uninterrupted reign:* A reign of a benign leader, who faithfully processes all requests, cannot be disrupted.

5.2.2 Client Alerts. Similar to conventional BFT systems, ENGRAFT leverages the clients to alert misbehavior of the leader. Specifically, if a client has not received responses from the leader within a time limit, it broadcasts the request to all nodes in the cluster, using a new `ClientAlert` RPC. The `ClientAlert` RPCs serve as alerts to benign nodes that the current leader may not behave normally.

Upon receiving a `ClientAlert` request, a follower first relays the request to the leader and then sets up a timer for malicious leader detection. If the leader is benign, it will broadcast the `ClientAlert` response to all followers and the client, to show that it has processed the client's regular request. If the current leader does not reply to the follower with a `ClientAlert` response before the timer expires, the follower enters the *preVote* stage. If at least $f + 1$ followers enter pre-voting, a new leader will be elected.

Note that in ENGRAFT, the request processing flow is done inside enclaves and hence the code integrity is guaranteed. The code of ENGRAFT specifies that the leader answers a `ClientAlert` RPC with a to-be-appended log containing the client request. As such, the attacker who operates the leader node only has two options: 1) reply to ClientAlert RPC and append the log; 2) refuse to reply to ClientAlert RPC and then lose the leader authority.

6 IMPLEMENTATION

This section presents some implementation details, including techniques for security enhancement and performance optimization.

6.1 BRaft Porting

We implemented ENGRAFT for Linux using the Open Enclave SDK (version: 0.17.0), on the top of BRaft (version: 1.1.1) [1], an open source implementation of the Raft protocol. There are three key components of BRaft:

- *Core Raft logic.* The core Raft logic implements the Raft consensus algorithm.
- *RPC framework.* The RPC framework is responsible for sending and receiving network packets, providing services to the core Raft logic code.
- *Thread management.* BRaft leverages coroutines atop pthread for efficient thread management. Each pthread worker manages a queue to store its assigned tasks (*i.e.*, coroutines) and keeps track of the execution context of every coroutine, including a local variable stack and CPU register values, to enable fast context switches between coroutines.

As components of BRaft are tightly coupled, we treat BRaft as a whole and port all three components into the enclaves, including the core Raft logic, the RPC framework, and the coroutine management. As such, the coroutine synchronization is rather efficient in ENGRAFT because coroutines are scheduled inside enclaves and no enclave transition occurs.

6.2 Security Enhancement

The implemenation of TIKS and MLD is straightforward. Here we only discuss other security enhancement in ENGRAFT.

6.2.1 File Encryption. The three persistent variables, *currentTerm*, *votedFor* and *log*, are stored in two files. The *currentTerm*, *votedFor* and the hash value of *log* are securely stored in TIKS. The mass log file is stored in an encrypted form outside enclaves. The encryption key is the SGX seal key and the encryption algorithm is 128-bit AES-GCM, which offers both confidentiality and integrity. Each log entry is encrypted separately. When appending a new log entry, ENGRAFT firstly encrypts it, then appends the ciphertext to the log data file, and finally updates the log hash summary. The freshness of log entries are protected by the log meta file stored in TIKS.

6.2.2 Remote Attestation. Nodes in ENGRAFT communicate with each other over Transport Layer Security (TLS) channels (v1.3). Especially, ENGRAFT integrates remote attestation evidence in a self-signed certificate and then exchanges this certificate during handshake [34]. In this way, every node conducts remote attestation when establishing TLS channels with others, to ensure (1) the remote party is running inside a genuine SGX enclave with expected security properties and (2) the remote party's enclave measurement is expected. In ENGRAFT, all nodes share the same enclave binary, so their measurements are the same. After finishing handshake and passing remote attestation, ENGRAFT successfully establishes an secure attested TLS channel.

6.3 Performance Optimizations

ENGRAFT is an I/O intensive system, with heavy network and storage operations. The overhead primarily comes from enclave transitions—the entering and exits of enclaves [63]. We perform the following notable optimization to reduce performance overhead.

Switchless OCALLs have been proposed by Weisse *et al.* [65], in which the caller threads inside the enclave place tasks in a shared memory buffer and worker threads outside pull the tasks from the buffer and execute them. Switchless operations have become increasingly prevalent for SGX programs demanding high performance. Several SGX SDKs, such as Intel SGX SDK and the Open Enclave SDK [9], have provided an option to utilize switchless operations for performance boost. These SDKs, however, merely offer general switchless interfaces. ENGRAFT further optimizes the most frequently used switchless operations for I/O operations and clock reads, which we detail below.

I/O queues. Fig. 4 shows an overview of the switchless framework in the design of ENGRAFT. There are two types of threads facilitating the switchless framework, namely worker threads in the host and receiver threads in the enclave. The number of the two types of threads can be adjusted as needed. ENGRAFT maintains two I/O queues shared between the enclave and the host: the request queue and the response queue. These two queues are allocated in advance to reduce the overhead of frequent buffer allocation. When caller threads issue I/O operations, they enqueue I/O jobs in the request queue and wait until it finishes. Host worker threads continuously poll jobs from the request queue, perform the actual system calls, and write the results back to the response queue. Once the enclave receiver threads successfully dequeue the results from the response queue, they immediately notify the corresponding caller threads in the enclave to process the return values.

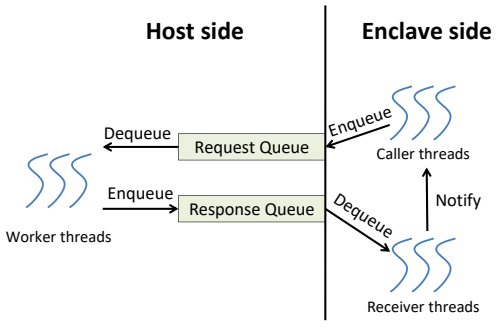


Figure 4: The workflow of ENGRAFT's switchless I/O.

Clock reads. ENGRAFT needs to frequently read the clock from the OS since its core logic relies on various timeout mechanisms. When the number of client connections increases, as ENGRAFT sets up a timer for each connection, the number of clock reads

also increases. Actually, the number of OCALLs for reading clocks is the highest among all enclave transitions in ENGRAFT. This is because the Raft protocol makes heavy use of timeouts (*e.g.*, RPC timeout and election timeout). To optimize clock reads, ENGRAFT creates a shared variable in the host to represent the clock readings and lets the worker threads in the host to update these variables after their finishing a batch of jobs (*e.g.*, when the request queue is empty). Once a caller thread needs to read clocks, it can directly read the shared variables instead of making OCALLs. Doing this does not impair the time precision much since the worker threads update the time value at a very high frequency. A read of this switchless clock takes only 2.2ns, which is 363× faster than Open Enclave's switchless clock reads (0.8μs) and is nearly 1800× faster than making an OCALL to read the clock (4μs).

7 EVALUATION

In evaluating the performance overhead of ENGRAFT, we mainly hope to answer the following questions:

- **Q1:** Compared to BRaft, how much performance overhead is introduced by SGX-related operations and various security countermeasures of ENGRAFT.
- **Q2:** Compared to similar BFT systems, how does ENGRAFT perform?

7.1 Experimental Setup

Test case. We deployed ENGRAFT to maintain a distributed counter, which can be read and increased by clients. Counter servers in the cluster execute on Intel CPUs with SGX features to run ENGRAFT, while clients are independent of ENGRAFT and hence they can be placed in any platforms. In this test case, clients interact with ENGRAFT via the `FetchAdd` RPC, and the request payloads are 128B in size. The leader that receives a `FetchAdd` request will commit a log in the cluster, apply the log (increase the counter) and finally reply to the client with the increased counter value. Following related studies, we only conduct performance evaluation of the stable phase in this test case, since unstable phase does not happen frequently and thus is less relevant to its overall performance.

Hardware and software specifications. We evaluate ENGRAFT in both LAN and WAN settings. In the LAN setting, we run ENGRAFT on cloud VM instances from the same data center. In the WAN setting, we use cloud VM instances from three data centers operated by the same cloud service provider. In both settings, each server in the cluster runs on one cloud instance. The client also runs on one of the cloud VMs. We configure multiple threads in the client instance to simulate the varying number of clients. All the aforementioned cloud VM instances are equipped with 8 vCPU (Intel Xeon Platinum 8369B) and run Ubuntu 20.04 with Linux kernel 5.4.0.

Batch processing. To increase throughput, we use batch processing technique with a batch size of 256. Note that batch processing does not introduce new rollback attack vectors since ENGRAFT can update the log meta file in a batch manner. If there is a crash during processing a batch of requests, the whole batch of requests is lost since the log meta file does not cover this batch of log entries.

Test protocols. We evaluate ENGRAFT with all security countermeasures (*e.g.*, TIKS and encrypted log storage) enabled, and we deploy the same counter cluster in the original BRaft implementation to examine the performance degradation and answer **Q1**. To compare with ENGRAFT, we also evaluate the performance of `Chained-Damysus` [25] in the same settings. `Chained-Damysus` is a HotStuff-based protocol that leverages SGX to improve crash resilience and reduce communication rounds. We note that `Chained-Damysus` does not provide confidentiality protection of its state machine as ENGRAFT does. We run a prototype of `Chained-Damysus` using its open-source code [2] without any modification of the server code.

7.2 Evaluation in LAN Settings

We first report the performance of ENGRAFT within 3-, 9-, 15-, 23-, and 31-node clusters. All nodes in these LAN clusters are virtually networked with a 5Gbps switch and the round-trip time (RTT) between any two VM instances is 0.05-0.07ms. Throughout the evaluation, we consider the `FetchAdd` RPC. Once receiving `FetchAdd` requests, the leader responds to the client after the cluster has successfully handled the requests. We treat finished `FetchAdd` requests as transactions and adopt transactions per second (TPS) as the throughput unit. We increase the request load by tuning the number of clients in the client VM and the rate of requests, and the latency and throughput are measured on the client side.

Fig. 5 shows the evaluation results. In the smallest 3-node cluster, BRaft's maximum throughput is 6.6× higher than that of ENGRAFT: ENGRAFT achieves 33 kTPS (with 9.54ms latency) while BRaft can reach 218 kTPS (with 6.72ms latency). This difference becomes bigger when the cluster scales up. In the 31-node cluster, BRaft achieves 91 kTPS, which is 17× higher than ENGRAFT (5.3 kTPS).

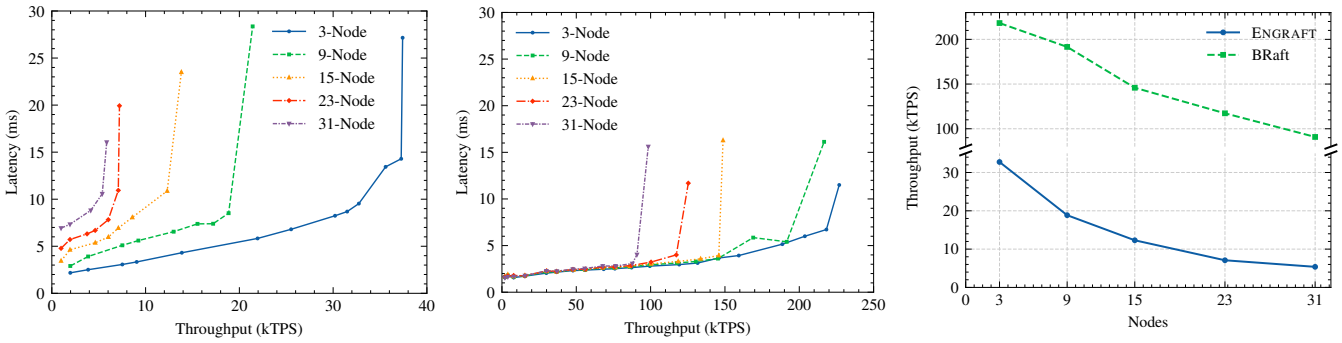
Fig. 5c shows that both ENGRAFT and BRaft exhibit obvious performance degradation when the cluster scales up in the LAN. For Braft, the maximum throughput achieved in the 31-node cluster is 42% of that in the 3-node cluster. As for ENGRAFT, this ratio is 16%. The worse scalability of ENGRAFT is expected since it runs inside enclaves and depends on TIKS for rollback prevention, which has a quadratic message complexity with respect to the number of participant nodes.

By comparing Fig. 5a and Fig. 5b, we can see that the latency of ENGRAFT increase faster than that of BRaft when the request load increases. This is due to enclaves' inefficiency in handling frequent network messages and TIKS's super-linear complexity.

7.3 Evaluation in WAN Settings

7.3.1 Test WAN Topology. To evaluate the performance of ENGRAFT in the WAN setting, we construct five network topologies, consisting of 5, 7, 21, 41 and 61 nodes, respectively. Nodes are evenly distributed among the three different data centers. The RTT between the first and the second data center is 11.3ms, while the one between the first and the third data center is 28.6ms. And the RTT between the second and the third data center is 27.2ms.

Moreover, to further understand the performance of the systems, we evaluate them in two network settings, *restricted network* and *unrestricted network*, with different bandwidth restrictions on the outgoing and incoming traffic of each VM. In the restricted



(a) ENGRAFT (b) BRaft (c) Maximum throughput vs. nodes

Figure 5: Latency vs. throughput in LAN.

network, which is the default setting, each VM is equipped with a 200Mbps network link; the bandwidth is throttled by the cloud provider. Observing that the peak bandwidth of the followers in the three systems (BRaft, ENGRAFT, and `Chained-Damysus`) is below 100Mbps while the peak bandwidth of the BRaft leader, the `Chained-Damysus` leader and the `Chained-Damysus` client can be much higher than 200Mbps, we then set up an unrestricted network to remove the bandwidth throttle imposed by the cloud provider to evaluate BRaft and `Chained-Damysus`. In the unrestricted network, follower VMs still use 200Mbps network links but the leader VM and the client VM are equipped with 2Gbps network links, respectively. Fig. 6 and Fig. 7a display the results in the unrestricted network, while Fig. 7b shows the maximum throughput of the systems in the restricted network.

7.3.2 Comparing with BRaft. As shown in Fig. 6a and Fig. 6b, when the cluster scales up, the latency of BRaft fluctuates slightly, but that of ENGRAFT increases, especially when the cluster grows to 41 or 61 nodes. In all these five WAN topologies, ENGRAFT handles requests at least 2× slower than BRaft. This is because TIKS requires two rounds of communication to ensure security, which introduces extra latency of two round-trip time compared with BRaft.

Fig. 7a illustrates that BRaft can achieve much larger throughput (*i.e.*, 4.5-25× larger) than ENGRAFT does when the network bandwidth is not a bottleneck. Meanwhile, Fig. 7b suggests that at a moderate bandwidth, the performance of ENGRAFT is comparable to that of BRaft in relatively larger clusters, namely those containing 21 nodes, 41 nodes and 61 nodes, respectively.

7.3.3 Comparing with `Chained-Damysus`. We address **Q2** by comparing ENGRAFT with `Chained-Damysus`. When evaluating `Chained-Damysus`, we also set the payload size to 128B and batch size to 256. By comparing Fig. 6a and Fig. 6c, we observe that both ENGRAFT and `Chained-Damysus` exhibit similar latency, but ENGRAFT has higher throughput with fewer nodes (*e.g.*, 1.8× higher in the 5-node cluster). However, Fig. 7b shows that the throughput of ENGRAFT drops below that of `Chained-Damysus` when the cluster contains more than 21 nodes. Moreover, `Chained-Damysus` has better scalability and its maximum throughput surpasses that of ENGRAFT in clusters with 41 or 61 nodes. The poor scalability of ENGRAFT in the WAN setting can be attributed to two reasons: First, ENGRAFT protects the states of replicas inside enclaves while `Chained-Damysus` only leverages SGX to generate and verify signatures, that is, ENGRAFT offers confidentiality to the state machine replicas while `Chained-Damysus` does not. Second, as will be seen shortly, the complexity of TIKS does not grow linearly.

It is worth pointing out that in the restricted network (see Fig. 7a) ENGRAFT always performs no worse than `Chained-Damysus`, demonstrating ENGRAFT is the better choice at moderate bandwidth.

7.4 Overhead Profiling

To better understand the causes of the overhead in ENGRAFT (per **Q1**), we implemented three different variants of ENGRAFT:

- Var-I. This variant replaces TIKS with simulated zero-overhead in-memory counters, eliminating the overhead due to TIKS.
- Var-II. This variant further removes log encryption described in Sec. 6.2.1 from Var-I, eliminating overhead due to file encryption.
- Var-III. This variant operates Var-II outside enclaves, eliminating overhead due to SGX.

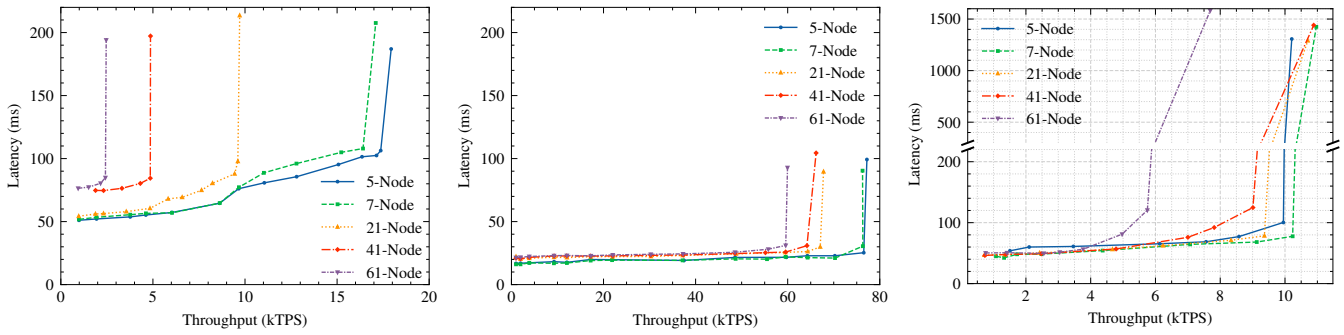
In the 5-node WAN cluster under the unrestricted network, the maximum throughputs and corresponding latencies of ENGRAFT, BRaft and the above three variants are listed in Table 4. We still consider `FetchAdd` RPC and use 128B payloads here.

Table 4: Overhead profiling for ENGRAFT.

Variant	Throughput (TPS)	Latency (ms)
ENGRAFT	17369	106.34
Var-I	30179	41.73
Var-II	31509	41.79
Var-III	76817	30.095
BRaft	76417	30.154

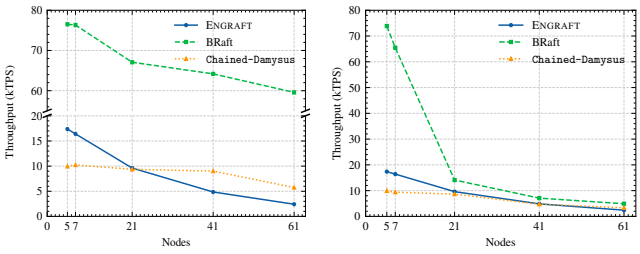
TIKS overhead. The gap between ENGRAFT and Var-I suggests that TIKS has high overhead in the WAN setting. ENGRAFT only reaches 58% of the throughput of Var-I.

Encryption overhead. The small difference between Var-I and Var-II suggests that using an encrypted log storage does not introduce much overhead. This is because, as measured in our tests, encrypting and decrypting 1KB data only takes 2.50μs and 10.54μs, respectively, which is small enough to be negligible in WAN.



(a) ENGRAFT (b) BRaft (c) Chained-Damysus

Figure 6: Latency vs. throughput in WAN (unrestricted network).



(a) Unrestricted network (b) Restricted network

Figure 7: Maximum throughput of the three systems.

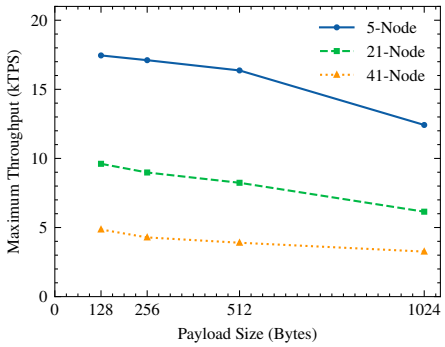


Figure 8: Maximum throughput vs. payload size

SGX overhead. Var-III reaches nearly the same maximum throughput as BRaft does, while Var-II only achieves 41% of the throughput of BRaft. This big gap suggests that the overhead of running BRaft in enclaves is dominant, even with the optimized switchless framework for ENGRAFT. There might be two remaining costs from SGX. First, as reported in [65], the overhead of accessing encrypted enclave memory could be expensive. Specifically, writing the enclave memory may bring 6.5-19.5% performance loss, while reading operations can result in 30-102% loss. Considering that ENGRAFT frequently accesses encrypted memory (*e.g.*, copying I/O buffers), the overhead cannot be neglected. Second, when enabling switchless OCALLs, ENGRAFT requires host worker threads to take extra CPU cores. In this test, ENGRAFT sets up two host worker threads to process OCALLS and thus its available cores were 25% less than BRaft on an 8 vCPU VM.

7.5 Impact of Payload Size

To further examine the performance of ENGRAFT, we evaluate ENGRAFT with larger payloads (when increased from 128B to 1024B) in three WAN topologies, *i.e.*, 5-, 21- and 41-node clusters. As shown in Fig. 8, ENGRAFT has a mild performance degradation when processing bigger requests. For instance, in the 5-node cluster, the maximum throughput with 1024B reaches 71% of that with 128B, while the payload is 4× larger.

8 DISCUSSION

Further optimization opportunities. Besides switchless OCALLs, recent studies utilize user space I/O library (*e.g.*, DPDK [3] and SPDK [12]) to optimize the performance of I/O intensive SGX systems [15, 16, 60]. This type of optimization still leverages the shared memory, but it maps DMA regions in the host memory. When enclave threads perform I/O operations, they directly interact with DMA regions mapped in the shared memory instead of putting jobs in the request queue. Compared with the existing switchless OCALL implementations, the userspace I/O optimization does not require extra host worker threads to poll jobs and thus saving CPU cores. We leave such optimization for ENGRAFT to future work.

Application scenarios. ENGRAFT can be used as a highly performant BFT protocol that supports federated blockchains or distributed clusters spanning multiple secure domains. Particularly, the WAN evaluation results in Sec. 7.3 suggest that ENGRAFT has good performance when deployed across multiple data centers. Moreover, ENGRAFT is also a BFT protocol that protects the confidentiality of the state replicated inside SGX enclaves. It allows all nodes to maintain a consistent encrypted log, such that no other parties can breach the secrecy of the log. One natural application of ENGRAFT is a confidential decentralized ledger, which accepts client requests in an encrypted form, and then decrypts the requests and performs updates or queries to the encrypted ledger via software in the enclaves.

Lessons learned. Our study suggests that porting CFT protocols (*e.g.*, Raft) inside TEEs (*e.g.*, SGX enclaves) does not directly offer BFT properties. This contradicts the presumptions made, either explicitly or implicitly, in some prior works [11, 15, 29, 54]. With model checking, we have identified a variety of vulnerabilities

that could breach the safety and liveness of Raft running inside SGX. When not using ENGRAFT, for instance, financial loss (*e.g.*, transactions revoked) may happen when a blockchain is built atop CCF [54].

Moreover, our study also serves as a wake-up call for the confidential computing community that build secure systems with TEEs: SGX, as well as other TEEs, are not bulletproof. Porting complex software into TEEs increases the TCB of the system. Software built with TEEs must also consider threats of memory safety [17, 42], side channels [22, 57, 61], state continuity [31], and concurrency bugs [64]. The large code base inside the TEEs would increase the likelihood of containing vulnerabilities that could violate the security assumption of TEEs. Therefore, to build secure confidential computing platforms using TEEs, one must find ways to balance the size of TCB and the richness of the functionalities.

Reconfiguration. Similar to related studies [20, 25, 32, 44, 62, 68], ENGRAFT does not yet consider dynamic reconfiguration of the network. However, the original Raft protocol introduces a joint consensus protocol to enable dynamic adjustment of network membership. As ENGRAFT is tightly intertwined with TIKS, however, doing so in ENGRAFT requires additional efforts to avoid security flaws. For instance, if the reconfiguration of ENGRAFT is not synchronized with that of TIKS, an outdated ENGRAFT node in the new configuration may send `Store` and `ConfirmStore` to TIKS nodes in the old configuration. We leave a comprehensive security analysis under a such scenario to future works.

9 RELATED WORK

Hardware-assisted Byzantine fault tolerance. Attested Append-only Memory (A2M) [24] is the first work to utilize a trusted log abstraction to defend against Byzantine behaviors; it can tolerate up to f arbitrary malicious nodes among $2f + 1$ nodes. Trusted incrementer (TrInc) [43] simplifies the trusted log abstraction to a trusted monotonic counter and achieves the same security guarantee. Based on the trusted counter, Veronese *et al.* [62] further introduce a signing function to significantly reduce overhead in traditional BFT protocols, *i.e.*, PBFT [20] and Zyzzyva [35]. CheapBFT [32] integrates three protocols for three typical scenarios, namely, the normal phase without attack, the switching phase after detecting malicious behaviors, and finally, the terminate phase running a BFT protocol [62]. FastBFT is a scalable variant of BFT protocol that leverages Intel SGX and secret sharing schemes [44] to achieve $O(n)$ communication complexity in the normal phase. Damysus [25] is a BFT protocol based on HotStuff [68]. It utilizes SGX as a trusted checker to increase Byzantine resilience and a trusted accumulator to reduce communication rounds.

In the above studies that leverage TEEs as the underlying trusted hardware [25, 32, 43, 44, 62], TEEs are primarily used to build small trusted components like trusted counters. The state machines are not protected by TEEs and hence their confidentiality is not guaranteed. In contrast, ENGRAFT protects the confidentiality of the state machines using SGX, enabling application scenarios that cannot be supported by these prior studies.

Confidential computing. TEEs provide confidentiality for running code and data and thus enable a new computing paradigm called confidential computing. Numerous studies have built confidential computing platforms for confidential map-reduce [56], secure distributed coordination of cloud applications [19], encrypted databases [52], anonymous communication networks [33] and confidential network middleware [26, 51].

Close to our work are those integrating blockchain frameworks with confidential computing. Specifically, Ekiden [23] is a privacy-preserving off-chain system that separates the whole blockchain architecture into compute nodes and consensus nodes. The compute nodes are a cluster of SGX-enabled machines used to process confidential data with smart contracts, while the consensus nodes are used to maintain a distributed ledger. CCF [54] is a framework for providing confidential services in permissioned blockchains by maintaining a distributed key-value store inside enclaves. CCF can be configured to run atop CFT or BFT consensus protocols, but it does not modify the underlying consensus algorithms much.

Rollback prevention for trusted hardware. Prior works have used non-volatile memory (NVRAM) to prevent rollback attacks [50, 58, 59]. Memoir [50] stores the chained hash of a sequence of states in the NVRAM of a TPM to prevent state rollback and leverages Uninterruptible Power Supply (UPS) to reduce the number of NVRAM writes. A similar idea has been taken by ICE [58] that reduces TPM writes using a dedicated capacitor hardware. Ariadne [59] encodes counters in a balanced Gray code form to maximize the durability of the TPM NVRAM. It also provides a protocol to achieve recoverability with rollback prevention.

Close to our design of TIKS is ROTE [45], which modifies an echo broadcast protocol to construct a distributed counter service inside SGX enclaves, achieving better performance without accessing NVRAM. Though the communication protocol of TIKS resembles the one in ROTE, these two schemes differ in the following aspects: First, ROTE assumes SGX may be compromised and so requires additional mechanisms, while the threat model of TIKS is identical with the one of ENGRAFT (as described in Sec. 3.2). Second, ROTE provides an abstraction of monotonic counters, while TIKS is a KV store for storing ENGRAFT meta files. Finally, compared to ROTE, TIKS additionally offers recoverability from crashes.

Formal analysis of Raft. Ongaro *et al.* [10] developed a formal specification of Raft using the TLA+ specification language [40] and an informal proof of safety for Raft. However, the formal specification only modeled the CFT environment, for example, servers crash and later restart from the persistent storage on disk, or the network may reorder, drop, and duplicate messages. In ENGRAFT, we specify the Byzantine fault model that is not contained in [10], such as the untrusted file system and the network. Wilcox et al. [66] proposed Verdi, which is a Coq-based framework for implementing and formally verifying state machine replication algorithms. An end-to-end proof of its safety is completed also under the Verdi framework (with 45000 additional lines of Coq code) [67]. However, the network semantics and fault models supported by Verdi do not include the Byzantine fault model we consider in ENGRAFT.

10 CONCLUSION

This paper presents ENGRAFT, a secure enclave-guarded Raft implementation that combines the best properties of a performant CFT protocol and a secure SGX enclave. ENGRAFT goes beyond

simply porting Raft into SGX enclaves, by performing a systematic analysis of this strawman solution and fixing both safety and liveness issues via improved protocol design and system implementation. The contributions of ENGRAFT are twofold: On the one hand, ENGRAFT achieves consensus on a cluster of $2f + 1$ machines tolerating up to f nodes exhibiting Byzantine-fault behavior. On the other hand, ENGRAFT allows the reuse of a production-quality Raft implementation in the development of a highly performant BFT protocol.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. Michael Reiter is supported in part by the NIFA Award 2021-67021-34252.

REFERENCES

[1] [n.d.]. BRaft. https://github.com/baidu/braft. (Accessed: 2022-05-03).
[2] [n.d.]. Damysus Source Code. https://github.com/vrahli/damysus. (Accessed: 2022-05-03).
[3] [n.d.]. Data Plane Development Kit (DPDK). https://www.dpdk.org. (Accessed: 2022-05-03).
[4] [n.d.]. ENGRAFT Source Code. https://github.com/wwl020/ENGRAFT. (Accessed: 2022-09-08).
[5] [n.d.]. The Intel Converged Security and Management Engine IOMMU Hardware Issue - CVE-2019-0090 and CVE-2020-0566. https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/cve-2019-0090-whitepaper.pdf. (Accessed: 2022-05-03).
[6] [n.d.]. Intel Product Specifications. https://ark.intel.com/content/www/us/en/ark/search/featurefilter.html. (Accessed: 2022-05-03).
[7] [n.d.]. INTEL-SA-00307: Intel CSME Advisory. https://www.intel.com/content/www/us/en/support/articles/000056085/software/chipset-software.html. (Accessed: 2022-05-03).
[8] [n.d.]. Intel Software Security Guidance. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/overview.html. (Accessed: 2022-05-03).
[9] [n.d.]. Open Enclave SDK. https://openenclave.io. (Accessed: 2022-05-03).
[10] [n.d.]. Safety proof and formal specification for Raft. http://raftuserstudy.s3-website-us-west1.amazonaws.com/proof.pdf. (Accessed: 2022-05-03).
[11] [n.d.]. Signal Secure Value Recovery. https://signal.org/blog/secure-value-recovery. (Accessed: 2022-05-03).
[12] [n.d.]. Storage Performance Development Kit (SPDK). https://spdk.io. (Accessed: 2022-05-03).
[13] [n.d.]. The support of trustworthy monotonic counters on SGX platforms. https://www.intel.com/content/www/us/en/support/articles/000057968/software/intel-security-products.html. (Accessed: 2022-05-03).
[14] [n.d.]. TiKV. https://tikv.org. (Accessed: 2022-05-03).
[15] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 65–79.
[16] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 173–190.
[17] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1213–1227. https://www.usenix.org/conference/usenixsecurity18/presentation/biondo
[18] Sean Braithwaite, Ethan Buchman, Igor Konnov, Zarko Milosevic, Ilina Stoilkovska, Josef Widder, and Anca Zamfir. 2020. Formal specification and model checking of the Tendermint blockchain synchronization protocol. In *2nd Workshop on Formal Methods for Blockchains*.
[19] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Proceedings of the 17th International Middleware Conference*. ACM, Trento Italy, 1–13. https://doi.org/10.1145/2988336.2988350
[20] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*. USENIX Association, New Orleans, LA. https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance
[21] Shanwei Cen and Bo Zhang. 2017. Trusted time and monotonic counters with intel software guard extensions platform services. *Online at: https://software. intel. com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services. pdf* (2017).
[22] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 142–157. https://doi.org/10.1109/EuroSP.2019.00020
[23] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 185–200.
[24] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-Only Memory: Making Adversaries Stick to Their Word. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 189–204. https://doi.org/10.1145/1323293.1294280
[25] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. 2022. DAMYSUS: Streamlined BFT Consensus Leveraging Trusted Components. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3492321.3519568
[26] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. 2019. LightBox: Full-Stack Protected Stateful Middlebox at Lightning Speed. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2351–2367. https://doi.org/10.1145/3319535.3339814
[27] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. 2004. Directed explicit-state model checking in the validation of communication protocols. *International journal on software tools for technology transfer* 5, 2 (2004), 247–267.
[28] Mark Ermolov and Maxim Goryachy. 2017. How to hack a turned-off computer, or running unsigned code in intel management engine. *Black Hat Europe* (2017).
[29] Mingyuan Gao, Hung Dang, and Ee-Chien Chang. 2021. TEEKAP: Self-Expiring Data Capsule Using Trusted Execution Environment. In *Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 235–247. https://doi.org/10.1145/3485832.3485919
[30] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (Tel-Aviv, Israel) *(HASP '13)*. New York, NY, USA, Article 11, 1 pages.
[31] Mohit Kumar Jangid, Guoxing Chen, Yinqian Zhang, and Zhiqiang Lin. 2021. Towards Formal Verification of State Continuity for Enclave Programs. In *30th USENIX Security Symposium*. USENIX Association, 573–590. https://www.usenix.org/conference/usenixsecurity21/presentation/jangid
[32] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-Efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. New York, NY, USA, 295–308. https://doi.org/10.1145/2168836.2168866
[33] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. 2017. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 145–161. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kim-seongmin
[34] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating remote attestation with transport layer security. *arXiv preprint arXiv:1801.05863* (2018).
[35] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 45–58.
[36] Vladimir Kukharenko, Kirill Ziborov, Rafael Sadykov, and Ruslan Rezin. 2021. Verification of hotstuff bft consensus protocol with TLA+/TLC in an industrial setting. In *Computer Science On-line Conference*. Springer, 77–95.
[37] L. Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 872–923.
[38] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227.279229
[39] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)* (December 2001), 51–58.
[40] Leslie Lamport. 2002. *Specifying systems*. Vol. 388. Addison-Wesley Boston.
[41] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* (July 1982), 382–401.
[42] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. 2017. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver,

BC, 523–539. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk
[43] Dave Levin, John (JD) Douceur, Jay Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (proceedings of the 6th usenix symposium on networked systems design and implementation (nsdi) ed.). 1–14.
[44] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. 2019. Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing. *IEEE Trans. Comput.* 68, 1 (2019), 139–151. https://doi.org/10.1109/TC.2018.2860009
[45] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*. 1289–1306.
[46] Stephan Merz. 2000. Model Checking: A Tutorial Overview. In *Summer School on Modeling and Verification of Parallel Processes*. Springer, 3–38.
[47] Diego Ongaro. 2014. *Consensus: Bridging theory and practice*. Stanford University.
[48] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 305–319.
[49] Bryan Parno. 2008. Bootstrapping Trust in a "Trusted" Platform. In *Proceedings of the 3rd Conference on Hot Topics in Security* (San Jose, CA) *(HOTSEC'08)*. USENIX Association, USA, Article 9, 6 pages.
[50] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. 2011. Memoir: Practical State Continuity for Protected Modules. In *2011 IEEE Symposium on Security and Privacy*. 379–394.
[51] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 201–216. https://www.usenix.org/conference/nsdi18/presentation/poddar
[52] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. Enclavedb: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.
[53] Michael K. Reiter. 1994. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security* (Fairfax, Virginia, USA) *(CCS '94)*. Association for Computing Machinery, New York, NY, USA, 68–80. https://doi.org/10.1145/191177.191194
[54] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cedric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olga Ohrimenko, Felix Schuster, Roy Schuster, Alex Shamis, Olga Vrousgou, and Christoph M Wintersteiger. [n.d.]. CCF: A Framework for Building Confidential Verifiable Replicated Services. ([n. d.]), 17.
[55] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167
[56] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 38–54.
[57] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 753–768. https://doi.org/10.1145/3319535.3354252
[58] Raoul Strackx, Bart Jacobs, and Frank Piessens. 2014. ICE: A Passive, High-Speed, State-Continuity Scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '14)*. New York, NY, USA, 106–115.
[59] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 875–892. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/strackx
[60] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. 2021. Rkt-Io: A Direct I/O Stack for Shielded Execution. In *Proceedings of the Sixteenth European Conference on Computer Systems*. ACM, Online Event United Kingdom, 490–506. https://doi.org/10.1145/3447786.3456255
[61] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 88–105.
[62] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (2013), 16–30. https://doi.org/10.1109/TC.2011.221
[63] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. Sgx-Perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Proceedings of the 19th International Middleware Conference*. ACM, Rennes France, 201–213. https://doi.org/10.1145/3274808.3274824
[64] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *Computer Security – ESORICS 2016*, Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows (Eds.). Springer International Publishing, Cham, 440–457.
[65] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, Toronto ON Canada, 81–93. https://doi.org/10.1145/3079856.3080208
[66] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958
[67] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol *(CPP 2016)*. New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081
[68] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591
[69] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 54–66.

A SAFETY PROPERTIES IN TLA+

- *Election safety:* There is at most one leader at each term.

$$ElectionSafety \triangleq \neg(\exists i, j \in Server : BothLeader(i, j))$$

$$\begin{aligned} BothLeader(i, j) \triangleq \wedge i \neq j \\ \wedge currrentTerm[i] = currentTerm[j] \\ \wedge state[i] = Leader \\ \wedge state[j] = Leader \end{aligned}$$

- *Log matching:* All log entries up to a log entry with the same index and term are identical in any two servers.

$$\begin{aligned} LogMatching \triangleq \forall i, j \in Server : \\ \forall n \in (1..Len(log[i])) \cap (1..Len(log[j])) : \\ log[i][n].term = log[j][n].term \Rightarrow \\ SubSeq(log[i], 1, n) = SubSeq(log[j], 1, n) \end{aligned}$$

- *Leader completeness:* Log entries that have been committed will appear in all leader's log entries with higher term.

$$\begin{aligned} LeaderCompleteness \triangleq \forall i, j \in Server : \\ state[i] = Leader \wedge currentTerm[i] \geq currentTerm[j] \Rightarrow \\ IsPrefix(Commited(j), log[i]) \end{aligned}$$

$$\begin{aligned} IsPrefix(xlog, ylog) \triangleq \wedge Len(xlog) \leq Len(ylog) \\ \wedge xlog = SubSeq(ylog, 1, Len(xlog)) \end{aligned}$$

$$Committed(i) \triangleq SubSeq(log[i], 1, commitIndex[i])$$

- *State machine safety:* If a state machine of one server have received a log entries at a index, all other servers will apply the same log entry in the same index.

$$\begin{aligned} StateMachineSafety \triangleq \forall i, j \in Server : \\ \vee \wedge commitIndex[i] \leq commitIndex[j] \\ \wedge SubSeq(log[i], 1, commitIndex[i]) \\ = SubSeq(log[j], 1, commitIndex[i]) \\ \vee \wedge commitIndex[i] > commitIndex[j] \\ \wedge SubSeq(log[i], 1, commitIndex[j]) \\ = SubSeq(log[j], 1, commitIndex[j]) \end{aligned}$$

B SAFETY AND LIVENESS

B.1 Safety of TIKS

We prove that a node can obtain the latest value tuple (in the form of $\langle index, value\rangle$ of each KV item $\langle key, \langle index, value\rangle\rangle$) maintained by TIKS, when recovering from crashes. We assume that a recovery operation (see Sec. 5.1.6) collects G (a set of value tuples with the largest index retrieved in "reconstruction" stage). The set G may contain one or more tuples corresponding to a specific key as there may be index conflict (see Sec. 5.1.6). Under index conflict, any two tuples in G share the same largest index (denoted as $index_l$), *i.e.*, $\forall\langle index_i, c\rangle, \langle index_j, d\rangle \in G, index_i = index_j = index_l$. The most recent successfully written value (*i.e.*, the value is stored) is considered as $\langle index_w, a\rangle$. We use t_a to represent the time when state a is generated.

Lemma B.1. *If an item $\langle index, value\rangle$ has been successfully written (i.e., completes Storage update (see Sec. 5.1.5)), there are at least $f+1$ of its copies in all nodes.*

Proof. In the storage update stage, at time t', the storage updating node passes the first round and the nodes that have responsed `Store` RPC requests are considered as R_1. Obviously, $|R_1| \geq f$. Let R_2 denotes the nodes that have the updated the item at t' and have responded `ConfirmStore` RPC requests, and thus $R_2 \subseteq R_1$. Since nodes may crash and lost storages in the first round, we can obtain $0 \leq |R_2| \leq |R_1|$. The updated item is successfully written only when at least f `ConfirmStore` responses are received (see Sec. 5.1.5), such that $|R_2| \geq f$. In the cluster, R_2 and the updating node maintain the written item, and thus at least $f+1$ nodes (including the updating node itself) have copies of the item. □

Lemma B.2. $index_w \leq index_l$.

Proof. We prove it by contradiction. If $index_w > index_l$, there are at least $f+1$ copies whose index is smaller than $index_w$ at timer t, which is conflicting with Lemma B.1 (*i.e.*, $index_w$ has been successfully written and at least $f+1$ nodes has its storage). Therefore, $index_w \leq index_l$. □

Lemma B.3. *If $index_w = index_l$, $\langle index_w, a\rangle \in G$.*

Proof. Assume $\langle index_w, a\rangle$ is successfully written at time t_1, thus at least $f+1$ nodes maintain its copies at this time (Lemma B.1). Let N represent the set of nodes with $\langle index_w, a\rangle$ at t_1 and N' represent the set of nodes responding to the inquiry request at recovery stage. Since $N + N' \geq f + 1 + f + 1 = 2f + 2$, we can obtain $N \cap N' = N'' \neq \emptyset$. For each node in N'', with the constraint $index_w = index_l$, its storage is $\langle index_w, a\rangle$. Thus, $\langle index_w, a\rangle \in G$. □

Lemma B.4. *If $index_w < index_l$, $\langle index_l, c\rangle \in G \wedge t_c > t_a$ holds (t_c means the time when c is generated).*

Proof. We prove it by contradiction. Since $index_l$ can be read, $index_l - 1$ has been successfully written. If an item $\langle index_i, d\rangle$ is generated after $\langle index_l, c\rangle$ (*i.e.*, $t_d > t_c$), we can get $index_i > index_l - 1$ and $index_i \geq index_l$. Thus, if $\exists\langle index_l, c\rangle \in G, t_a \geq t_c$, it indicates $index_w \geq index_l$, conflicting with $index_w < index_l$. As such, If $index_w < index_l$, $\langle index_l, c\rangle \in G \wedge t_c > t_a$. □

Lemma B.5. *$\langle index_w, a\rangle \in G \vee (\langle index_l, c\rangle \in G \wedge (index_l > index_w) \wedge (t_c > t_a))$ is true.*

Proof. The lemma can be derived directly from Lemma B.2 , Lemma B.3, as well as Lemma B.4. □

Theorem B.6. *A TIKS node restores to only an up-to-date state in recovery.*

Proof. We prove this holds in all cases, *i.e.*, with and without index conflict.

No index conflict. If there are no conflicts, there will be only one tuple, say, $\langle index_r, b\rangle$ in G. Following Lemma B.5, we know $\langle index_w, a\rangle = \langle index_r, b\rangle \vee (index_r > index_w \wedge t_b > t_a)$ holds. This means that the recovering node either reads stored states or newer states that have not been successfully stored, both of which are regarded as "up-to-date states" in ENGRAFT.

Under index conflict. If there are conflicts, the recovering node consults with the leader (we assume its term is $term''$) for up-to-date states following the description in Sec. 5.1.6. Next we show that the node can restore up-to-date persistent variables, *i.e.*, $currentTerm$, $votedFor$ and log storage.

First, the recovering node can obtain the highest term value (denoted as $term'$) appeared in one or multiple conflicting Raft meta files. And $term'' \geq term'$ holds if it later discovers a leader it will consult with. Clearly, by setting $currentTerm$ to $term''$, the node's $currentTerm$ is up-to-date after recovery.

Second, the reset of $votedFor$ does not breach Raft's "one vote per a term" principle since $votedFor$ already points to the leader at $term''$ and the node will not vote for others at this term. This implies that $votedFor$ cannot be rolled back.

Third, Lemma B.10 states that the ENGRAFT leader has the up-to-date committed logs, and thus the recovering node can obtain up-to-date logs by synchronizing with the leader. □

B.2 Liveness of TIKS

We prove TIKS has liveness by showing that a correct TIKS node can always finish the storage update or recovery.

Lemma B.7. *A correct TIKS node can always finish the storage update.*

Proof. Following the storage update algorithm (Sec. 5.1.5), a node updating its key-value items should communicate with f other nodes, which in total requires at least $f+1$ correct nodes

to finish. Since our threat model specifies there are at least $f + 1$ correct nodes in the cluster at any time, a correct node can always successfully finish the update. □

LEMMA B.8. *A correct TIKS node can always finish the storage recovery.*

PROOF. For storage recovery (Sec. 5.1.6), a recovering node needs to communicate with other $f + 1$ nodes, which is always available per our threat model because recovering nodes is deemed faulty. Note that the additional requirement of the Raft leader's participation in handling index conflict does not halt the recovery as $f + 1$ correct nodes eventually elect a leader in ENGRAFT. □

THEOREM B.9. *TIKS has liveness.*

PROOF. According to Lemma B.7 and Lemma B.8, a correct TIKS node can always finish the storage update or recovery. As such, TIKS has liveness. □

B.3 Safety of ENGRAFT

We first prove a lemma used in the safety proof of TIKS (Theorem B.6 depends on Lemma B.10) and then prove the safety of ENGRAFT.

LEMMA B.10. *The ENGRAFT leader always has up-to-date committed logs.*

PROOF. We prove by induction:

Base case. At the very beginning of $term_0$, every initialized node has the same log storage and so does the elected leader L_0. Throughout $term_0$, L_0 always appends client requests earlier than other nodes per Raft specification. As such, this lemma holds for $term_0$.

Induction step. Assume this lemma holds for $term_i$. Then the leader L_i has the newest committed logs. Without loss of generality, we denote the last committed log entry at $term_i$ as log_i.

We use N and $node_k$ to denote all and one of the nodes that have successfully appended log_i (*i.e.*, have persisted log_i and updated the log meta in TIKS) at $term_i$. $node_k$ keeps log_i when it does not crash. There are two cases when $node_k$ crashes.

(1) $node_k$ crashes, recovers, encounters no TIKS index conflict and restores its log storage containing log_i per TIKS property (Lemma B.5).
(2) $node_k$ encounters index conflict, restores $term_i$ as described in Sec. 5.1.6 and consults with a leader. First, it turns to L_i and retrieves log_i if L_i is still in power. Second, L_i may no longer be a leader and $node_k$ will finally consult with a leader L_x at $term_x$, where $term_x > term_i$. Since a recovering node is deemed crashed and cannot vote, the election of L_x has nothing to do with $node_k$ and so does the one of L_{i+1}.

Given the above observation, we demonstrate that the quorum Q that elects L_{i+1} as the leader must have log_i. Since Q and N are two quorums, we have $|N'| = |Q \cap N| \geq 1$. Bacically, N' only consists of two types of nodes: 1) nodes that do not crash; 2) nodes that crash but recover log_i regardless of index conflicts. As such, the quorum that elects L_{i+1} as the leader must have log_i, and thus L_{i+1} will contain log_i per Raft's election specification. Again, L_{i+1} always appends client requests earlier than other nodes throughout $term_{i+1}$, *i.e.*, the lemma holds for $term_{i+1}$.

The proof of the induction step is complete.

Conclusion. By the principle of induction, the ENGRAFT leader always has the up-to-date committed logs. □

THEOREM B.11. *ENGRAFT provides safety.*

PROOF. First, Raft provides safety on crash-faulty nodes. Our model checking results (see Sec. 4.2) reveal the safety violations of putting Raft inside TEEs with Byzantine faulty hosts. To prevent these violations shown in Table 3, we propose three kinds of countermeasures: file encryption that prohibits arbitrary modification on persistent files, rollback prevention that resists state rollback as well as network encryption and authentication that establishes trusted communication channels between nodes. ENGRAFT provides rollback prevention according to Theorem B.6.

With the three countermeasures, we tackle safety violations of porting Raft to enclaves where host OSs can exhibit arbitrary faults. In other words, ENGRAFT provides safety on Byzantine faulty nodes with enclaves just as what Raft does on nodes that solely exhibit crash faults. □

B.4 Liveness of ENGRAFT

As described in Sec. 5.2.1, there are three liveness requirements, namely misbehavior exposure, election of a benign leader, and uninterrupted reign. Raft relies on two of the mechanisms to provide liveness: a timeout mechanism to detect a defunct leader and trigger a random election and periodic heartbeats to maintain the leader's authority. In ENGRAFT, we integrate MLD (see Sec. 5.2) to guarantee liveness when encountering Byzantine behaviors. We next show how the aformentioned `ClientAlert` can be combined with Raft's two existing mechanisms to offer liveness to ENGRAFT.

LEMMA B.12. *ENGRAFT satisfies the misbehavior exposure property.*

PROOF. Suspicious leaders are reported by the client to all replicas, who will collect evidence of such malicious behavior by relaying the request and timing the leader's response. As such, a malicious leader who does not respond to the client's `ClientAlert` requests will be evident to the $f + 1$ benign nodes, who will trigger a new leader election. Thus, the misbehavior exposure property holds. □

LEMMA B.13. *ENGRAFT satisfies the uninterrupted reign property.*

PROOF. `ClientAlert` does not break the uninterrupted reign property. As the benign leader will send heartbeats or `ClientAlert` responses to followers in time, the benign nodes will not enter *preVote* phase. Therefore, a malicious node cannot collect $f + 1$ *preVotes* to become a candidate (thus a leader), *i.e.*, the uninterrupted reign property holds. □

As in related works [20, 44], a BFT protocol can achieve liveness (once the system is synchronous) if it ensures misbehavior exposure, election of a benign leader, and uninterrupted reign. Lemma B.12 and Lemma B.13 ensure the first and the last of these properties. Election of a benign leader within $f+1$ leadership changes could be enforced as in PBFT [20], i.e., by electing leaders in a round-robin fashion. Raft, however, does not do so, and so liveness is contingent on Raft's method of electing a leader eventually installing a benign one.