Formally Verifying a Rollback-Prevention Protocol for TEEs

Weili Wang[1], Jianyu Niu[1], Michael K. Reiter[2], and Yinqian Zhang[1(✉)]

[1] Research Institute of Trustworthy Autonomous Systems and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
12032870@mail.sustech.edu.cn, niujy@sustech.edu.cn, yinqianz@acm.org
[2] Duke University, Durham, NC, USA
michael.reiter@duke.edu

Abstract. Formal verification of distributed protocols is challenging and usually requires great human effort. Ivy, a state-of-the-art formal verification tool for modeling and verifying distributed protocols, automates this tedious process by leveraging a decidable fragment of first-order logic. Observing the successful adoption of Ivy for verifying consensus protocols, we examine its practicality in verifying rollback-prevention protocols for Trusted Execution Environments (TEEs). TEEs suffer from rollback attacks, which can revert confidential applications' states to stale ones to compromise security. Recently, designing distributed protocols to prevent rollback attacks has attracted significant attention. However, the lack of formal verification of these protocols leaves them potentially vulnerable to security breaches. In this paper, we leverage Ivy to formally verify a rollback-prevention protocol, namely the TIKS protocol in ENGRAFT (Wang *et al.*, CCS 2022). We select TIKS because it is similar to other rollback-prevention protocols and is self-contained. We detail the verification process of using Ivy to prove a rollback-prevention protocol, present lessons learned from this exploration, and release the proof code to facilitate future research (https://github.com/wwl020/TIKS-Proof-in-Ivy). To the best of our knowledge, this is the first endeavor to explain the formal verification of a rollback-prevention protocol in detail.

Keywords: Formal verification · Rollback attacks · Trusted execution environments (TEEs)

1 Introduction

The intricate nature of distributed systems makes it challenging to prove their correctness. Errors in manual proofs [2] of well-known consensus protocols have already demonstrated the difficulty of this reasoning. Given this challenge, researchers tend to leverage formal verification tools such as Coq [1], TLAPS [3], and Dafny [23], to develop more reliable machine-checked proofs. Unfortunately, applying these tools to distributed protocols usually requires tremendous human

Published by Springer Nature Switzerland AG 2024
V. Castiglioni and A. Francalanza (Eds.): FORTE 2024, LNCS 14678, pp. 155–173, 2024.
https://doi.org/10.1007/978-3-031-62645-6_9

effort. For instance, the Raft proof [38] written in Coq contains approximately 50,000 lines of code (a proof-to-code ratio of 10), and the development of a state machine replication library (IronFleet [13]) in Dafny took 3.7 person-years. This daunting proof effort greatly limits the practicality of these verification tools.

Padon *et al.* [28] Ivy, a new verification tool that uses decidable logic. Protocol designers need to express their protocols in a decidable fragment and figure out inductive invariants to prove the safety of the protocols. To ease this tedious, error-prone process, Ivy will showcase counterexamples to decidability and invariance. Once the protocol is well expressed, Ivy can highly automate the verification process by generating verification conditions and leveraging the SMT solver to check their satisfiability. In this way, Ivy sacrifices the protocol description's expressiveness to reduce the verification process's complexity. Witnessing the success of Ivy in verifying distributed protocols [8,27,35] with much less human effort, we were curious whether Ivy could be used to verify rollback-prevention protocols that ensure state continuity in Trusted Execution Environments (TEEs).

TEEs such as Intel SGX [14], AMD SEV [5] and ARM CCA [7], have revolutionized the field of confidential computing, enabling the execution of sensitive operations within secure and isolated environments called enclaves. TEEs provide a hardware-based foundation for confidential computation, safeguarding the confidentiality and integrity of sensitive data. However, the presence of rollback vulnerabilities can compromise the security of TEEs and render them susceptible to unauthorized access or manipulation. In a rollback attack, an adversary rolls back the application's state to a previous one, thereby potentially bypassing security measures or gaining unauthorized control. The impact of a successful rollback attack can be severe in real-world scenarios. For example, rollback attacks can break the safety of a TEE-guarded consensus protocol [36].

To address this issue, various rollback-prevention systems have recently been proposed. Prominent examples include ROTE [25], TIKS [36], Narrator [26] TEEMS [11], and Narrator-Pro [29], which, despite their innovative approaches to rollback prevention, have not undergone rigorous formal verification for their security claims. Although the core protocol of Nimble [6], a recently proposed rollback-prevention system, has been claimed to be formally verified, there are few explanations of the verification process, which leaves a gap in understanding the verification.

In this paper, we present the formal verification of TIKS (**T**rustworthy distributed **I**n-memory **K**ey-value **S**torage), a rollback-prevention protocol by Wang *et al.* [36], using Ivy. We provide comprehensive details and insights into the specific techniques and methodologies employed in formally verifying TIKS. We study the formal verification of TIKS for several reasons. A series of rollback-prevention protocols, including ROTE [25], Narrator [26] and TIKS [36], share a common core, *i.e.*, a customized echo broadcast protocol [30]. Both TIKS and Narrator improve ROTE by directly storing states instead of monotonic counters. However, TIKS is self-contained and does not rely on external components, whereas Narrator relies on a blockchain. Therefore, TIKS is a more suitable

candidate to represent these rollback-prevention protocols, and its verification can serve as a foundation for verifying other protocols. In verifying TIKS, we specifically aim to address the following research questions.

- **R1**: Is there a difference between the verification of rollback-prevention protocols and other distributed protocols?
- **R2**: What type of simplification is necessary to facilitate the verification of rollback-prevention protocols?
- **R3**: What is the user experience of using Ivy to verify rollback-prevention protocols?

We expressed TIKS in Ivy and attempted to verify it directly. There are two highly correlated challenges in this process. First, Ivy's decidability restriction limits the expressiveness of the protocol description. Second, finding proper invariants to eliminate spurious counterexamples becomes more difficult under the decidability restriction. To address these challenges, we developed reasonable simplifications to the TIKS protocol. First, we simplified the state retrieval by making the recovering node directly read other node's states. Second, we reversed the recovery steps to defer the recovering node's state reconstruction. With these simplifications, we obtained a proof that does not lie inside the decidable fragment but generates solvable verification conditions and thus we can verify it in Ivy. As TIKS shares similar core components with other rollback-prevention protocols [25,26], we believe that our proof strategy can be generalized to other rollback-prevention protocols to facilitate their formal verification.

Contributions. Our contributions are summarized below:

- We present a step-by-step verification of the TIKS protocol in Ivy. As far as we know, this is the first endeavor to formally verify a rollback protocol in Ivy, with detailed explanation of such procedure.
- We showcase the essential protocol simplifications that make the protocol verifiable in Ivy, without affecting the core workflow of the protocol. This strategy can be generalized to other rollback-prevention protocols.
- We demonstrate the practicality of Ivy in verifying rollback-prevention protocols, paving the way for the formal verification of other similar systems.

2 Background

2.1 Rollback Attacks on TEEs

A rollback attack occurs when an adversary reverts the execution of an application by rolling back its state to a previous version. For instance, the enclave first persists state s_1 (at time t_1), followed by the state s_2 (at time t_2). At a later time t_3, the enclave needs to retrieve the newest state, possibly due to a crash. However, since the adversary controls the external storage, it can manipulate the system and provide the enclave with a stale state s_1, without being detected. As

a result, the victim enclave will load s_1 and revert its execution to s_1 instead of continuing from s_2.

In reality, rollback attacks pose a significant threat to many confidential applications. Consider the case of password guessing. To prevent brute-force attacks, an application will record the number of failed login attempts and lock the account when the number exceeds a threshold. However, if the adversary can roll back the state of the application, it can bypass the protection mechanism by reverting the number of failed login attempts to a previous value, thus conducting brute-force attacks without being detected.

2.2 Formal Verification Tools

Coq [1] is an interactive theorem prover that allows users to use higher-order logic reasoning to build proofs. TLAPS [3] is a theorem prover designed for TLA+ [20], a formal specification language for modeling and validating programs. Dafny [23] is a verification-aware imperative language allowing users to write Hoare-style protocol invariants and verify the correctness with Z3 solver [4].

The aforementioned verification tools generally require great human effort. To deal with this issue, Padon *et al.* [28] leverage a decidable fragment of first-order logic to build Ivy, a verification framework that highly automates the verification process. Ivy users first express their protocols in pure first-logic and then restrict protocol descriptions to a decidable fragment. In this process, Ivy will constantly showcase counterexamples to decidability, helping users to refine protocol descriptions. After that, Ivy will generate verification conditions and leverage its SMT solver (Z3) to check the correctness. Since the generated verification conditions are in the decidable fragment, they can be discharged by the solver in finite time and thus provide timely feedback to the designers. In contrast, other tools leveraging SMT solvers (*e.g.*, Dafny) may generate undecidable verification conditions and may take an infinite amount of time to solve.

2.3 Preliminary Knowledge of Ivy

Decidable Logic. Ivy adopts Effectively Propositional Logic (EPR), a decidable fragment of first-order logic, to represent distributed protocols. A logic formula is an EPR fragment if it has a prenex normal form of $\exists X_1, X_2, ..., X_N$, $\forall Y_1, Y_2, ..., Y_M. P(X_1, ..., X_N, Y_1, ..., Y_M)$, where the predicate P does not contain function symbols except the acyclic (stratified) ones. Acyclic function symbols are functions that do not have cycles in their definition. For instance, a function that maps from sort s_1 to sort s_2 and a function that maps from sort s_2 to sort s_1 are considered cyclic, and therefore not permitted in EPR.

Ivy requires users to express their protocols in EPR fragment and then Ivy will generate decidable verification conditions and leverage its SMT solver (Z3) to check their satisfiability. However, successful proofs in Ivy do not necessarily lie in the decidable fragment. Ivy supports undecidable logic fragments to express complex protocols that are not easy to represent in EPR, although undecidable

verification conditions burden the SMT solver and may take an infinite amount of time to solve.

Actions. Ivy models protocol procedures as isolated actions that operate on protocol states. Similar to functions in conventional programming languages, Ivy actions also have input arguments. Actions can be invoked by other actions or the environment. Ivy uses the "export" keyword to mark an action exported, which then can be called by the environment. Although an action can be invoked inside another action, Ivy does not support recursive action calls. To simplify the concurrency reasoning, actions execute atomically without any interruption. In other words, Ivy uses the interleaving model to model concurrency—the environment can call actions in an arbitrary order and thus sequentially executed actions actually model arbitrary interleaving of protocol procedures.

Inductive Invariants. To specify the safety property of a protocol, users have the option to write assertions in the form of preconditions (using the "require" keyword) and postconditions (using the "ensure" keyword) within an action. However, in a more general sense, users should construct inductive invariants that imply the safety of the protocol being verified. These invariants must hold after the initialization phase. Moreover, Ivy checks whether the invariants hold at the beginning of an action and whether they are preserved after the execution.

3 TIKS Recap

TIKS establishes a distributed in-memory key-value (KV) storage abstraction that provides rollback protection. Each TIKS node maintains one in-memory KV store and a TIKS cluster consists of $2f + 1$ nodes, where f is the maximum number of nodes hosted on malicious hosts. Malicious nodes can drop, duplicate, and reorder network messages, as well as crash. States requiring freshness guarantees (*i.e.*, crucial states that cannot be rolled back) are stored in TIKS as key-value items. For a KV item $\langle key, value \rangle$, *key* uniquely identifies the state, whereas *value* contains the state and a monotonic index indicating the version. As such, *value* is represented as a 2-tuple: $\langle index, state \rangle$. The *key* of a state usually begins with the ID of the node that owns the state and ends with the state name. For example, when node A stores its states named s_1 and s_2 in TIKS, the keys of these states are A_s_1 and A_s_2, respectively.

To offer rollback protection, *i.e.*, ensuring successfully stored KV items will not be overwritten by older versions, TIKS involves a storage update sub-protocol and a storage recovery sub-protocol. We briefly introduce these two sub-protocols below and refer the readers to its original description [36] for more details.

3.1 Storage Update

TIKS use a two-round communication process to enable a node to persist states with freshness in the cluster. Whenever a node wants to update a KV item,

it first updates its own KV store and then issues `Store` and `ConfirmStore` remote procedure calls (RPCs) to other nodes to update their KV stores. Figure 1 illustrates this workflow. In this three-node cluster, Node A successfully updates its KV store after collecting f `Store` responses and f `ConfirmStore` responses from other nodes.

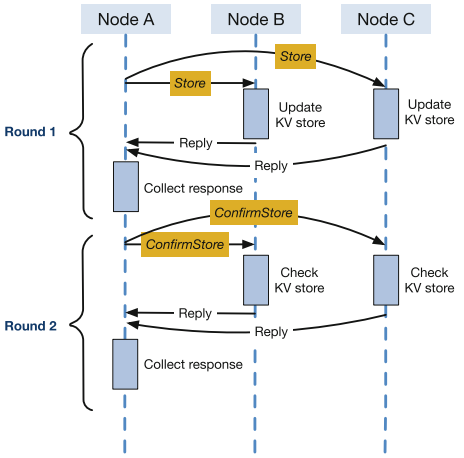

Fig. 1. Workflow of storage update in TIKS.

The First Round. The updating node first updates the KV item in its KV store, and then broadcasts `Store` RPCs. A `Store` request encapsulates the key of the to-be-updated state and its value. A benign node receiving such request will update its KV store accordingly if the index of the received state is larger than the index of the one in its own KV store, and then reply with a boolean value to indicate the update status. With at least f responses, there is a majority of nodes (including the updating node itself) that have successfully updated their KV stores, and thus the updating node passes the first round. Otherwise, the updating node continues to broadcast `Store` RPCs until it collects at least f responses from other nodes.

The Second Round. After passing the first round, the updating node then issues `ConfirmStore` requests with the same content as the `Store` requests, to the nodes having responded to the first-round requests. When receiving a `ConfirmStore` request, a benign node will reply with a success value if it meets two conditions. First, it must have responded to the corresponding `Store` request. Second, its KV store must contain an item that is identical to the one in the `ConfirmStore` request. If these conditions are not met, the benign node will simply ignore the request. If the updating node collects at least f responses in the second round, it can ensure that the state has been securely stored in TIKS. Otherwise, the updating node fails to update the state and must retry.

3.2 Storage Recovery

A TIKS node maintains an in-memory KV store, and thus it needs to recover its KV store from others after crashes, following steps below.

Inquiry. The crashed node broadcasts `RetrieveStorage` requests until it has successfully collected at least $f + 1$ responses from other nodes. A benign node receiving such a request will reply with all KV items in its KV store.

Reconstruction. By collecting $f+1$ KV stores from the alive nodes, the crashed node can recover the newest KV items. Specifically, the crashed node first reconstructs its own KV store. As TIKS uses a monotonic index to indicate the version of a state, the crashed node picks the one with the largest index among the received KV stores for each KV item. Then, the crashed node writes its own states back to the cluster using `Store` and `ConfirmStore` RPCs.

The use of a monotonic index in an alive node's aborted storage update requests may cause trouble in its later recovery as one index may be used by multiple states, which is termed as an index conflict. For instance, a node that has successfully stored $state_6$ at index 6 may crash during the process of storing the newer version (denoted as $state_7$) at index 7. In the recovery, the node may not observe $state_7$ and restore $state_6$ as $state_7$ has not been successfully stored. Later, the node may crash again when storing a different version (denoted as $state'_7$) at index 7. In this case, during recovery, the node may observe $state_7$ and $state'_7$ with the same index 7 and cannot decide which one to keep. By design, TIKS does not handle index conflicts and defers the resolution to the application layer. The application layer can randomly choose the state with the largest index or adopt other strategies to resolve conflicts. For example, ENGRAFT [36] uses the Raft leader to decide which state to keep.

4 Proof

A TIKS KV item has a structure of $\langle key, \langle index, state \rangle \rangle$. Inside the value tuple, *index* tracks the version of *state*, and the state with the largest index is the latest. As explained in Sect. 3.2, standalone TIKS protocol does not handle index conflicts and leaves it to the application layer. Consequently, the *state* field is irrelevant to rollback reasoning and we remove it in the proof. Moreover, storing nodes' multiple states (these states have different keys) in one KV store is equivalent to storing their states in multiple separate KV stores, each of which contains states with the same key. As such, we assume that each node only has one state and uses its node ID as the key. A KV item in the proof has this structure: $\langle nodeID, index \rangle$.

TIKS nodes hosted on malicious hosts cannot deviate from the protocol, but they still can drop, duplicate, and reorder network messages, as well as crash at any time. These threats are modeled in the proof.

Next, we report the attempt of using Ivy to formally prove the safety property of TIKS, *i.e.*, a TIKS node can obtain its own state ($\langle index \rangle$) maintained in the TIKS cluster when recovering from crashes.

4.1 Types and Functions

Uninterpreted Types. We do not need to represent the exact meaning of different types. For instance, we just need a type declaration of nodes when reasoning protocol participants. We use *node*, *quorum*, *index*, and *nonce* types.

The node type is used to represent a single TIKS node, while the quorum type is used to represent a majority of nodes (*i.e.*, at least $f + 1$ nodes). As Ivy does not support the mix of arithmetic reasoning and EPR, we cannot directly define the quorum type by checking the cardinality of node sets. Instead, the quorum concept in Ivy is modeled using an axiom that indicates quorum intersection [27]. First, we use a relation to reason whether a node is in a quorum: `relation member (N:node, Q:quorum)` (identifiers beginning with capital letters like "N" and "Q" are placeholders in Ivy). When member (n1, q1) is true, node n1 is in the quorum q1. Second, we define an axiom to reason quorum intersection: `axiom forall Q1:quorum, Q2:quorum. exists N:node. member (N, Q1) & member(N, Q2)`. With this reasoning, the quorum type can be used to represent a majority of nodes.

The index type is used to represent the index of a KV item $\langle nodeID, index \rangle$. In the implementation, indexes should be interpreted as integers. However, we do not need this integer interpretation and related arithmetic operations in the proof—we just need to ensure that we can compare two indexes (*i.e.*, the total order property). Ivy's built-in order library provides a module that defines unbounded sequence (0 is the minimum value), which basically establishes a total order on the type. As such, we define the index type as an instantiation of this module: `instance index: unbounded_sequence`.

Similar to the index type, the nonce type is also an instantiation of the unbounded sequence module. In our implementation, a node recovered from a crash will re-establish TLS connections with other nodes in the cluster, which enables the recovered node to distinguish its own messages sent from previous incarnations. To model this in the proof, we assign each node a nonce that is incremented after each recovery and make each sent message contain the sender's nonce.

Defined Functions. Ivy functions can be evaluated on arguments to produce deterministic results, and we use them to model partial states of a node. We define three functions to represent states that only have one value per node: n_committed_index records the largest index that a node has successfully written; n_recovered_index records the largest index that a node has successfully retrieved from the cluster in recovery; and n_nonce records the nonce of a node.

4.2 Relations

In the proof, we use relations to model KV stores, network messages, and crashes.

KV-Store Representation. To represent a node's KV store, we define: `relation n_tiks_states (N0:node, N:node, I:index)`. For example, when

node n0 stores the KV item $\langle n1, i1 \rangle$ for node n1, n_tiks_states(n0,n1,i1) should be true.

Network Messages. As listed in Listing 1.1, m_store_req and m_store_resp relations denote the requests and responses of `Store` RPC. For example, when node n0 with nonce no0 wants to update its index to i1 and broadcasts `Store` requests, m_store_req(n0,no0,i1) will be set to true; when node n1 with nonce no1 receives the request and replies to node n0, m_store_resp(n1,no1,n0,no0,i1) will be set to true. Similarly, the m_confirm_store_req and m_confirm_store_resp relations represent the requests and responses of `ConfirmStore` RPC. The `RetrieveStorage` RPC is modeled using m_recover_req and m_recover_resp relations. m_recover_req(n0,no0) is set to true when the recovering node n0 with nonce no0 broadcasts `RetrieveStorage` requests. An alive node n1 with nonce no1 replying to the request will set m_recover_resp(n1,no1,n0,no0,n,i) to true if it contains an item $\langle n, i \rangle$ in its KV store (*i.e.*, n_tiks_states(n1,n,i) is true).

The above relation-based network reasoning follows the approach of the Paxos proof written in Ivy [27] and models an abstracted network layer that allows message dropping, duplication, and reordering.

Listing 1.1. Relation-based network messages representation

```
# Sender, sender's nonce, index
relation m_store_req(S:node, NO:nonce, I:index)
# Sender, sender's nonce, dest, dest's nonce, index
relation m_store_resp(S:node, SNO:nonce, D:node, DNO:nonce, I:index)
# Sender, sender's nonce, index
relation m_confirm_store_req(S:node, NO:nonce, I:index)
# Sender, sender's nonce, dest, dest's nonce, index
relation m_confirm_store_resp(S:node, SNO:nonce, D:node, DNO:nonce, I:index)
# Sender, sender's nonce
relation m_recover_req(S:node, NO:nonce)
# Sender, sender's nonce, dest, dest's nonce, (N, I) means n_tiks_states(S, N, I)
    ↪ holds
relation m_recover_resp(S:node, SNO:nonce, D:node, DNO:nonce, N:node, I:index)
```

Crash Representation. We use a relation to record node crashes: `relation crash(N:node)`. If node n0 is crashed, crash(n0) is set to true. To ensure the presence of a majority of alive nodes, we then define an inductive invariant: `invariant ~(exists Q:quorum. forall N:node. member(N, Q) -> crash(N))`. Similarly, we use `relation recovering(N:node)` to record whether a node is recovering. A recovering node must be crashed, and thus we have: `invariant forall N:node. recovering(N) -> crash(N)`.

4.3 Actions

Initialization. In the verification, Ivy first checks whether a program's initialization satisfies the inductive invariants. This initialization is defined using the `after init` keyword. The initialization action mainly sets relations to false as there are no messages and crashed nodes in the beginning.

Two-Round Communication. The first round involves the send_store action (Listing 1.2) that broadcasts `Store` requests, and the reply_store action (Listing 1.3) that processes a `Store` request. The second round involves the send_confirm_store action (Listing 1.4) broadcasting `ConfirmStore` requests, the reply_confirm_store action (Listing 1.5) handling a `ConfirmStore` request, and the store_success action (Listing 1.6) that updates a node's written index. The common precondition of these actions is that the node is not crashed.

send_store action. The preconditions require the to-be-written index is larger than the node's largest successfully written index appearing in `written_index`. The node first updates its KV store (Line 4 in Listing 1.2), broadcasts `Store` requests (Line 5) and finally replies to itself (Line 6).

Listing 1.2. The send_store action

```
1  action send_store(n:node, i:index) = {
2      require ~crash(n);
3      require i > n_committed_index(n);
4      n_tiks_states(n, n, i) := true;
5      m_store_req(n, n_nonce(n), i) := true;
6      m_store_resp(n, n_nonce(n), n, n_nonce(n), i) := true;
7  }
```

reply_store action. The preconditions require the requesting node has broadcasted `Store` requests. The responder updates its KV store per the request (Line 5 in Listing 1.3, recall that the capital letter "I" means a placeholder). In this update, the responder only updates the KV store if the index is larger than the current index (`if I <= i`); otherwise, it keeps the current state (`else n_tiks_states(n0, n, I)`). After updating the KV store, the responder sends replies to the requester (Line 6).

Listing 1.3. The reply_store action

```
1  action reply_store(n0:node, n:node, i:index) = {
2      require ~crash(n0);
3      require exists NO:nonce. m_store_req(n, NO, i);
4      require n0 ~= n;
5      n_tiks_states(n0, n, I) := true if I <= i else n_tiks_states(n0, n, I);
6      m_store_resp(n0, n_nonce(n0), n, n_nonce(n), i):= true;
7  }
```

send_confirm_store action. Before broadcasting `ConfirmStore` requests, the node in this action first ensures that it has received a majority of `Store` responses. We enforce this requirement via a precondition (Line 8 in Listing 1.4).

Listing 1.4. The send_confirm_store action

```
1  action send_confirm_store(n:node, i:index, q:quorum) = {
2      require ~crash(n);
3      require n_tiks_states(n, n, i);
4      require i > n_committed_index(n);
5      require m_store_req(n, n_nonce(n), i);
6      require m_store_resp(n, n_nonce(n), n, n_nonce(n), i);
7      # Pass the first round
8      require forall N:node. member(N, q) -> m_store_resp(N, n_nonce(N), n, n_nonce(n), i
           ↪ );
9      m_confirm_store_req(n, n_nonce(n), i) := true;
10     m_confirm_store_resp(n, n_nonce(n), n, n_nonce(n), i) := true;
11 }
```

reply_confirm_store action. The preconditions not only require the requesting node has broadcasted `ConfirmStore` requests but also ensure that the receiving node has stored the to-be-written index and responded to the `Store` RPC (Line 4 in Listing 1.5). The receiving node then replies to the requester (Line 6).

Listing 1.5. The reply_confirm_store action

```
1  action reply_confirm_store(n0:node, n:node, i:index) = {
2      require ~crash(n);
3      require exists NO:nonce. m_store_req(n, NO, i) & m_confirm_store_req(n,NO,i);
4      require m_store_resp(n0, n_nonce(n0), n, n_nonce(n),i);
5      require n0 ~= n;
6      m_confirm_store_resp(n0, n_nonce(n0), n, n_nonce(n), i) := true if n_tiks_states(n0
           ↪ , n, i) else m_confirm_store_resp(n0, n_nonce(n0), n, n_nonce(n), i);
7  }
```

store_success action. The preconditions require the node has received a majority of `ConfirmStore` responses. The node then updates its written index.

Listing 1.6. The store_success action

```
1  action store_success(n:node, i:index, q:quorum) = {
2      require ~crash(n);
3      require i > n_committed_index(n);
4      require n_tiks_states(n, n, i);
5      require m_store_req(n, n_nonce(n), i) & m_confirm_store_req(n, n_nonce(n), i);
6      # Pass the first and second round
7      require forall N:node. member(N, q) -> m_confirm_store_resp(N, n_nonce(N), n,
           ↪ n_nonce(n), i) & m_store_resp(N, n_nonce(N), n, n_nonce(n), i);
8      n_committed_index(n) := i;
9      n_recovered_index(n) := i;
10 }
```

Crash Modeling. We model the crash and recovery using eight actions. These actions include one action for crashing the node (node_crash) and seven actions for the recovery process (nonce_increase, send_recover_req, send_recover_resp, node_recover, rec_send_store, rec_send_confirm_store, and rec_store_success). The nonce_increase action and the rec_send_store action are only invoked in recovery actions and thus are not exported.

The nonce_increase action increases the recovering node's nonce. It first updates the nonce, and then clears all messages corresponding to the new nonce.

In the crash action shown in Listing 1.7, the precondition requires that only alive nodes can be crashed. A node is marked as crashed (`crash(n) := true`) if its crash does not result in a quorum with all crashed nodes. Recall that we model a set of at least $f+1$ nodes as a quorum type, and thus a quorum with all crashed nodes means there are at least $f+1$ crashed nodes, which contradicts the threat model that at most f nodes can crash.

Listing 1.7. The node_crash action

```
1  action node_crash(n:node) = {
2      require ~crash(n);
3      crash(n) := true;
4      if exists Q:quorum. forall N:node. member(N, Q) -> crash(N) {
5          crash(n) := false;
6      } else { }
7  }
```

The recovery begins with the send_recover_req action (Listing 1.8) where the recovering node increases its nonce and broadcasts `RetrieveStorage` requests. Alive nodes receiving such requests will reply with their KV stores as shown in the send_recover_resp action (Listing 1.9).

Listing 1.8. The send_recover_req action and

```
action send_recover_req(n:node) = {
    require crash(n);
    call nonce_increase(n);
    recovering(n) := true;
    m_recover_req(n, n_nonce(n)) := true;
}
```

Listing 1.9. The send_recover_resp action

```
action send_recover_resp(n0:node, n:node) = {
    require ~crash(n0);
    require m_recover_req(n, n_nonce(n));
    require n0 ~= n;
    m_recover_resp(n0, n_nonce(n0), n, n_nonce(n), N, I) := true if n_tiks_states(n0, N
        ↪ , I) else n_tiks_states(n0, N, I);
}
```

Listing 1.10 displays the node_recover action. A recovering node that has collected a majority of `RetrieveStorage` RPC responses will reconstruct its KV store and start writing back its index by invoking the rec_send_store action. The rollback-prevention is represented as a postcondition stating that a recovered node will retrieve an index not smaller than its written index right before the crash. We omit the rec_send_store, rec_send_confirm_store, and rec_store_success actions as they are similar to actions used to write an index via the two-round communication. In the rec_store_success action, the recovering node finishes the write-back operation and finally finishes the recovery.

Listing 1.10. The node_recover action

```
action node_recover(n:node, q:quorum, retrieved_i:index)={
    require crash(n) & recovering(n);
    require m_recover_req(n, n_nonce(n));
    require forall N:node. ~member(n, q) & member(N, q) -> ~crash(N);
    # Receive responses from the quorum
    require forall N:node. member(N, q) -> exists I:index. m_recover_resp(N,n_nonce(N),
        ↪  n,n_nonce(n),N,I);
    require forall S:node, I:index. m_recover_resp(S, n_nonce(S), n, n_nonce(n), n, I)
        ↪ -> I <= retrieved_i;
    require exists N:node. member(N, q) & m_recover_resp(N, n_nonce(N), n, n_nonce(n),
        ↪ n, retrieved_i);
    # Check rollback prevention
    ensure n_committed_index(n) <= retrieved_i;
    n_recovered_index(n) := retrieved_i;
    # KV-store reconstruction
    n_tiks_states(n, N, I) := false;
    n_tiks_states(n, N, I) := true if (exists N0:node. member(N0, q) & m_recover_resp(
        ↪ N0, n_nonce(N0), n, n_nonce(n), N, I)) else n_tiks_states(n,N,I);
    n_tiks_states(n, n, I) := true if I <= retrieved_i else n_tiks_states(n,n,I);
    call rec_send_store(n);
}
```

4.4 Inductive Invariants

Listing 1.11 lists all used invariants in the proof. Most of them are described before or self-explanatory and thus we only explain the invariant with "safety" notation. Recall that we aim to prove that a TIKS node can obtain its own state ($\langle index \rangle$) maintained in the TIKS cluster when recovering from crashes. The safety invariant depicts this property: if an updating node N successfully writes its index I to the cluster and there is a quorum Q that does not consist of crashed nodes, then in quorum Q, there exists a node N0 storing the index I for node N. Note that a recovering node can retrieve states from a quorum containing alive nodes only, and thus the safety invariant actually states that the recovering node can obtain its own state from the quorum.

Listing 1.11. Inductive invariants

```
1  invariant ~(exists Q:quorum. forall N:node. member(N, Q) -> crash(N))
2  invariant [safety] forall N:node, Q:quorum. exists N1:node. ~(exists N0:node. member(N0
       ↪ , Q) & crash(N0)) ->  member(N1, Q) & n_tiks_states(N1, N, n_committed_index(N
       ↪ ))
3  invariant m_confirm_store_resp(S, n_nonce(S), D, n_nonce(D), I) -> m_store_resp(S,
       ↪ n_nonce(S),D,n_nonce(D),I)
4  invariant m_store_resp(S, n_nonce(S), D, n_nonce(D), I) -> n_tiks_states(S, D, I)
5  invariant n_recovered_index(N) >= n_committed_index(N)
6  invariant forall I:index, N:node. I = n_committed_index(N) -> n_tiks_states(N,N,I)
7  invariant recovering(N) -> crash(N)
8  invariant recovering(N) & D ~= N -> ~m_store_resp(N, n_nonce(N), D, n_nonce(D), I)
9  invariant recovering(N) & I ~= n_recovered_index(N) -> ~m_store_resp(N, n_nonce(N), N,
       ↪ n_nonce(N), I)
10 invariant recovering(N) & D ~= N -> ~m_confirm_store_resp(N, n_nonce(N), D, n_nonce(D),
       ↪  I)
```

5 Protocol Simplifications

A successful Ivy proof mandates solvable verification conditions that can be solved by an SMT solver in finite time, and inductive invariants that imply the safety property. Without correct inductive invariants restricting the reachable states, Ivy will report spuriously unsafe states as counterexamples and fail the proof. In verification practice, finding inductive invariants for distributed protocols is a tedious and error-prone process [10,12,39]. We also encounter this challenge in the verification of the TIKS protocol, and Ivy's use of decidable logic fragment makes it even harder to represent the invariants—writing complex invariants usually leads to undecidable verification conditions.

Actually, making the proof decidable is the most challenging task throughout the verification. Protocol description and invariants easily introduce quantifier alternations [28] that result in undecidable verification conditions. Although Ivy developers propose approaches such as "derived relation" [27] and "relational abstraction" [34] to mitigate this issue, we find that their adoption in TIKS protocol is not straightforward and we cannot find a way to make the proof decidable. Fortunately, undecidable verification conditions are not necessarily unsolvable. We then turn to seek proper simplifications on the protocol to make the verification conditions solvable while preserving the core workflow of TIKS.

5.1 Simplified State Retrieval

Modeling the `RetrieveStorage` using m_recover_req and m_recover_resp relations fails the proof as the `RetrieveStorage` responses represented by m_recover_resp may not capture the KV store of the responding node at the time when it receives the `RetrieveStorage` request. We introduce the counterexample reported by Ivy below. Node 0 with the committed index 1 and nonce 1 is recovering (*i.e.*, m_recover_req(0,1) is true), and an alive node 1 with nonce 0 receives the `RetrieveStorage` request from node 0. Although node 1 has stored index 1 for node 0 (*i.e.*, n_tiks_states(1,0,1) is true), it does not reply to node 0 with this information (*i.e.*, m_recover_resp(1,0,0,1,0,1) is false). As such, in the node_recover action, node 0 cannot retrieve its latest index 1 and the proof fails. Using an invariant establishing the correspondence between m_recover_resp and n_tiks_states relations may help to eliminate this spurious counterexample. Ideally, this invariant should state that the KV store in a `RetrieveStorage` response is the same as the KV store of the responding node at the time when it receives the request. However, this reasoning involves storing KV-store histories of nodes, which makes the proof more complex and unverifiable in Ivy.

To facilitate the proof, we then simplify the recovery process by removing the `RetrieveStorage` RPC—a recovering node directly reads a majority of nodes' KV stores and then reconstructs its own KV store without broadcasting `RetrieveStorage` RPCs. As such, relations (m_recover_req and m_recover_resp) and actions (send_recover_resp and node_recover) related to the `RetrieveStorage` RPC are removed, and the node_recover_resp action is modified to directly read the KV stores of the responding nodes.

5.2 Reversed Recover Steps

In TIKS recovery, KV-store reconstruction and write-back operation are two separate steps and the finish of the latter step marks the end of recovery. A faithful modeling of this "reconstruct then write back" process yields a bogus counterexample. The counterexample shows that a recovering node having reconstructed its KV store can "forget" the reconstruction result at the time when it finishes the write-back operation, resulting a recovered node with stale KV store. Finding an invariant stating the "reconstruct then write back" order helps to eliminate this bogus counterexample but doing this in an undecidable logic fragment is more challenging, as the underlying SMT solver may be stuck for a long time. In practice, we simplify this process by reversing the order of the reconstruction and write-back steps.

As the write-back operation involves separate actions that conduct two-round communication, it is impossible to model the "reconstruct then write back" process in a single action. Instead, we reverse the order of these two steps, *i.e.*, following the "write back then reconstruct" order. When a recovering node successfully writes its retrieved index back, it then reconstructs its KV store from a majority of nodes. As such, the finish of the write-back operation and store

reconstruction can occur atomically and the invariant stating the order of these two steps is no longer necessary.

This reversed order does not deviate from the original TIKS protocol much, as putting reconstruction at the end of recovery also ensures that the recovered node has the latest KV store.

5.3 Proof Result

The above simplifications eases the invariant finding task while still preserving the core workflow of the TIKS protocol. Although the resulted proof does not fit in the decidable fragment of Ivy, its verification conditions can be discharged by Z3 in a reasonable time. It is important to note that in Ivy, a proof is considered successful regardless of whether it lies within the decidable fragment. Undecidable proofs can be established as long as the SMT solver is able to solve the verification conditions within a finite amount of time.

On a machine with Intel Core i7-10700 CPU and 32 GB RAM, Ivy takes 30 s to verify the protocol. In comparison, Ivy takes 2.8 s to verify the Multi-Paxos protocol [35]. The increased verification time is expected. First, the TIKS proof involves crash and recovery modeling that is not present in the Multi-Paxos proof and thus is more complex. Second, the TIKS proof involves undecidable verification conditions that bring more workload to the SMT solver.

6 Lessons Learned

In the verification, we obtain the following answers to our research questions:

- **R1 Answer**: Verification techniques from other distributed protocols can be applied to verify TIKS, such as relation-based message modeling and quorum reasoning. However, the verification of TIKS introduces new challenges, including modeling the recovery algorithm and representing the rollback-prevention property. These complexities make the proof construction lie outside the decidable fragment of Ivy. Our proof provides insights into the differences between verifying rollback-prevention protocols and other distributed protocols in Ivy.
- **R2 Answer**: We simplify the recovery process of TIKS to facilitate the verification, by making the recovering node directly read other node's states and deferring the recovering node's state reconstruction. Our exploration demonstrates that careful simplifications will not affect the core workflow of the protocol and can facilitate the verification greatly.
- **R3 Answer**: In verifying TIKS, we find that writing a proof in Ivy is straightforward, but making the proof decidable and establishing proper invariants are challenging. Although Ivy offers an interactive way (*i.e.*, reporting counterexamples) for users to find invariants, it is not easy to use in practice. First, expressing the protocol in a decidable fragment not only results in much human effort but also makes the proof less intuitive, which is not user-friendly.

Second, a user that turns to undecidable logic to express protocol actions and invariants cannot receive timely feedback from Ivy to refine their invariants, and thus may spend a great amount of time in finding proper invariants. From our experience, using Ivy does not necessarily make the proof easier than previous approaches. Although users write much less code in Ivy, they have to spend much more time in making the proof decidable with proper invariants.

7 Related Work

Formal Verification of Distributed Systems. Numerous works leverage formal verification tools introduced in Sect. 2.2 to verify distributed systems. Verdi [37] is a Coq-based framework for implementing and verifying state machine replication algorithms like Raft [37,38]. Disel is another Coq-based framework that allows the implementation and safety verification of distributed systems in a modular way [32]. TLAPS has been widely used to prove the safety of distributed protocols including Paxos protocol [9,22], Raft reconfiguration protocol [31] and Byzantine protocols [16,21]. IronFleet [13] is a Dafny-based verification framework that divides a distributed system into three verification layers and constructs proofs in a bottom-up manner using refinement. Ivy has been used to prove multiple Paxos variants and verify their implementation [27,35].

Parameterized model checking, a technique to check a system with arbitrary system size, can also be used in distributed system verification [17,19]. For instance, ByMC [19] models threshold-guarded distributed protocols as threshold automata and verifies their correctness in a counter system [18], which depicts a set of identical protocol participants.

Despite these efforts to formally verify distributed protocols, there has been a lack of focus on rollback-prevention protocols. Our work fills this gap and provides a detailed formal verification of a rollback-prevention protocol.

Formal Methods Used in TEEs. Jangid *et al.* [15] leverage Tamarin prover [33], a symbolic verification tool, to verify the rollback-prevention property of TEE applications by approximating the application in the execution logic of Tamarin. Wang *et al.* [36] use model checking to find vulnerabilities in an in-enclave crash fault-tolerant consensus protocol. They enumerate attack vectors in the TLC model checker [40] and use it to detect error traces. Li *et al.* [24] verify the firmware of ARM CCA in Coq. They build a layered verification framework and prove that the firmware implementation refines the top-level specification. The above works do not explore the formal verification of rollback-prevention protocols for TEEs, which is the focus of our work.

8 Conclusion and Future Work

This paper presents the formal verification for a rollback-prevention protocol, TIKS. We report the step-by-step verification process and showcase the user

experience of using Ivy in verifying rollback-prevention protocols, providing valuable insights for researchers and developers interested in verifying such protocols.

In future work, we plan to explore the possibility of eliminating the protocol simplifications by finding necessary invariants manually or automatically [12,39], and making the proof decidable by decomposing the proof [35]. Furthermore, it will be interesting to verify the liveness property of the TIKS protocol using Ivy.

Acknowledgments. Michael Reiter was supported in part by NIFA Award 2021-67021-34252. Yinqian Zhang was in part supported by National Key R&D Program of China (No. 2023YFB4503900). Jianyu Niu was supported in part by the NSFC under Grant 62302204.

References

1. The Coq proof assistant. https://coq.inria.fr. Accessed 03 May 2022
2. Errors found in distributed protocols. https://github.com/dranov/protocol-bugs-list. Accessed 03 May 2022
3. TLA+ proof system (TLAPS). http://tla.msr-inria.inria.fr/tlaps/content/Home.html. Accessed 03 May 2022
4. Z3 SMT solver. https://github.com/Z3Prover/z3. Accessed 03 May 2022
5. AMD secure encrypted virtualization. https://www.amd.com/en/processors/amd-secure-encrypted-virtualization
6. Angel, S., et al.: Nimble: rollback protection for confidential cloud services. In: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2023), pp. 193–208 (2023)
7. ARM confidential compute architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture
8. Berkovits, I., Lazić, M., Losa, G., Padon, O., Shoham, S.: Verification of threshold-based distributed algorithms by decomposition to decidable logics. In: Dillig, I., Tasiran, S. (eds.) CAV 2019, Part II. LNCS, vol. 11562, pp. 245–266. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_15
9. Chand, S., Liu, Y.A., Stoller, S.D.: Formal verification of multi-paxos for distributed consensus. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 119–136. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_8
10. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. Form. Methods Syst. Des. **49**, 190–218 (2016)
11. Dinis, B., Druschel, P., Rodrigues, R.: RR: a fault model for efficient tee replication. In: The Network and Distributed System Security Symposium. Internet Society (2023)
12. Hance, T., Heule, M., Martins, R., Parno, B.: Finding invariants of distributed systems: it's a small (enough) world after all. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2021), pp. 115–131 (2021)
13. Hawblitzel, C., et al.: IronFleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 1–17 (2015)
14. Intel software guard extensions. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html

15. Jangid, M.K., Chen, G., Zhang, Y., Lin, Z.: Towards formal verification of state continuity for enclave programs. In: 30th USENIX Security Symposium (USENIX Security 2021), pp. 573–590 (2021)
16. Jehl, L.: Formal verification of HotStuff. In: Peters, K., Willemse, T.A.C. (eds.) FORTE 2021. LNCS, vol. 12719, pp. 197–204. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78089-0_13
17. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: 2013 Formal Methods in Computer-Aided Design, pp. 201–209. IEEE (2013)
18. Konnov, I., Veith, H., Widder, J.: On the completeness of bounded model checking for threshold-based distributed algorithms: reachability. Inf. Comput. **252**, 95–109 (2017)
19. Konnov, I., Widder, J.: ByMC: byzantine model checker. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11246, pp. 327–342. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03424-5_22
20. Lamport, L.: Specifying Systems, vol. 388. Addison-Wesley, Boston (2002)
21. Lamport, L.: Byzantizing Paxos by refinement. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 211–224. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24100-0_22
22. Lamport, L., Merz, S., Doligez, D.: TLAPS proof of basic PAXOS. https://github.com/tlaplus/tlapm/blob/main/examples/paxos/Paxos.tla. Accessed 03 May 2022
23. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
24. Li, X., et al.: Design and verification of the arm confidential compute architecture. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022), pp. 465–484 (2022)
25. Matetic, S., et al.: ROTE: rollback protection for trusted execution. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 1289–1306 (2017)
26. Niu, J., Peng, W., Zhang, X., Zhang, Y.: Narrator: secure and practical state continuity for trusted execution in the cloud. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 2385–2399 (2022)
27. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. Proc. ACM Programm. Lang. **1**(OOPSLA), 1–31 (2017)
28. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 614–630 (2016)
29. Peng, W., Li, X., Niu, J., Zhang, X., Zhang, Y.: Ensuring state continuity for confidential computing: a blockchain-based approach. IEEE Trans. Depend. Secure Comput., 1–14 (2024). https://doi.org/10.1109/TDSC.2024.3381973
30. Reiter, M.K.: Secure agreement protocols: Reliable and atomic group multicast in rampart. In: Proceedings of the 2nd ACM Conference on Computer and Communications Security, CCS 1994, pp. 68–80. Association for Computing Machinery, New York (1994). https://doi.org/10.1145/191177.191194
31. Schultz, W., Dardik, I., Tripakis, S.: Formal verification of a distributed dynamic reconfiguration protocol. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 143–152 (2022)

32. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. Proc. ACM Programm. Lang. **2**(POPL), 1–30 (2017)
33. Tamarin prover. https://tamarin-prover.com/
34. Tamir, O., et al.: Counterexample driven quantifier instantiations with applications to distributed protocols. Proc. ACM Programm. Lang. **7**(OOPSLA2), 1878–1904 (2023)
35. Taube, M., et al.: Modularity for decidability of deductive verification with applications to distributed systems. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 662–677 (2018)
36. Wang, W., Deng, S., Niu, J., Reiter, M.K., Zhang, Y.: ENGRAFT: enclave-guarded raft on byzantine faulty nodes. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 2841–2855 (2022)
37. Wilcox, J.R., et al.: Verdi: a framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, New York, NY, USA, pp. 357–368 (2015). https://doi.org/10.1145/2737924.2737958
38. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.: Planning for change in a formal verification of the raft consensus protocol. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, pp. 154–165 (2016)
39. Yao, J., Tao, R., Gu, R., Nieh, J.: DuoAI: fast, automated inference of inductive invariants for verifying distributed protocols. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022), pp. 485–501 (2022)
40. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6