Securing Aptos Framework with Formal Verification

Junkil Park ✉
Aptos Labs, Santa Clara, CA, USA

Teng Zhang ✉
Aptos Labs, Santa Clara, CA, USA

Wolfgang Grieskamp ✉
Aptos Labs, Santa Clara, CA, USA

Meng Xu ✉
University of Waterloo, Canada

Gerardo Di Giacomo ✉
Aptos Labs, Santa Clara, CA, USA

Kundu Chen ✉
MoveBit, Hong Kong

Yi Lu ✉
Bitslab, Singapore, Singapore

Robert Chen ✉
OtterSec, USA

Abstract

The Aptos Framework is a collection of smart contracts written in the Move language that define standard and common on-chain actions for the Aptos Network. As the security and safety of the Aptos Framework is of utmost importance, it has continuously undergone rigorous testing and comprehensive auditing. To further increase the level of assurance, we have formally verified its security and correctness. This involves identifying critical security requirements for each module, creating formal specifications, and subsequently verifying them using the Move Prover. To the best of our knowledge, this represents one of the first instances of formal verification being applied on such a large scale in a smart contract framework. This paper discusses how this rigorous effort ensures a high level of quality assurance for the Aptos Framework.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Formal verification, Smart contracts, Aptos Network, The Move language, The Move Prover

Digital Object Identifier 10.4230/OASIcs.FMBC.2024.9

1 Introduction

The Aptos Network [1] is a safe, scalable, and upgradeable layer-1 blockchain with built-in support for the Move language designed for fast and secure transaction execution. The Aptos Framework[1], similar to an operating system for a computer, serves as the foundational platform for the Aptos Network, defining its core functionalities, managing on-chain resources, and offering a standardized environment for the development of user smart contracts. It comprises a suite of Move smart contract modules that define standard and common on-chain actions for the Aptos Network, such as prologue and epilogue of transactions, the staking mechanism, and Aptos Digital Asset Standard. It is imperative to ensure the correctness and security of the Aptos Framework because the unexpected behavior of such foundational Move modules could cause substantial asset loss or disrupt the normal functioning of the network. For this reason, the Aptos Framework has continuously undergone rigorous testing and comprehensive auditing. To further increase the level of assurance, we have formally verified its security and correctness against formal specifications derived from our comprehensive and systematic methodology. We identified critical security requirements for each module and created formal specifications for large parts of the Aptos Framework, which were then verified with the Move Prover.[2]

[1] `https://github.com/aptos-labs/aptos-core/tree/fmbc-24/aptos-move/framework`
[2] Notice that a preliminary summary of this paper has been published on Medium [2]

© Junkil Park, Teng Zhang, Wolfgang Grieskamp, Meng Xu, Gerardo Di Giacomo, Kundu Chen, Yi Lu, and Robert Chen;
licensed under Creative Commons License CC-BY 4.0
5th International Workshop on Formal Methods for Blockchains (FMBC 2024).
Editors: Bruno Bernardo and Diego Marmsoler; Article No. 9; pp. 9:1–9:16

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This rigorous approach has significantly enhanced the security and correctness assurance of the Aptos Framework. For instance, `block_prologue` is a critical Move function since it is executed at the beginning of each block. More importantly, `block_prologue` should never abort because any abort will effectively halt the network by preventing it from generating new blocks. As part of the liveness assurance of Aptos Network, we formally proved the absence of abort in `block_prologue`, despite that this function involves complex execution across 96 Move functions in 22 different Move modules. During this process, we identified and fixed some potential arithmetic overflows that may potentially trigger an abort in `block_prologue` (e.g., the `calculate_rewards_amount` function in the `stake` module).

Moreover, specifications form an integral part of the Aptos Framework *documentation*, which is in fact automatically generated based on the specifications and hence, provides an unambiguous and detailed account of the expected behavior for each function and module. Finally, integrating the Move Prover into the Continuous Integration (CI) process significantly reduces the time and cost associated with code auditing – once the specifications are pinned down, problematic code changes will likely trigger verification failure in the CI before manual auditing happens.

This paper will explain how we have secured the Aptos Framework through formal verification using the Move Prover. Section 2 introduces the Move language, Move Prover, and Aptos Framework. In Section 3, we will explain our methodology to verification of the Aptos Framework. Section 4 will discuss important findings and lessons learned from this effort. After describing the related work in Section 5, we will conclude in Section 6.

2 Background

2.1 Move as A Programming Language for Smart Contracts

The Aptos Network natively supports Move as its smart contract language. A Move program is essentially a sequence of updates that try to evolve a *global persistent memory state*, which we just call the *(global) memory*. Similar to other blockchains, updates are a series of atomic transactions. All runtime errors result in a transaction abort, which does not change the blockchain state except to transfer some currency ("gas") from the account that sent the transaction to pay for the cost of executing the transaction.

The global memory is organized as a collection of resources described by Move structures (i.e., data types). A resource in memory is indexed by a ⟨type, address⟩ pair. An address is a unique identifier in the Aptos Network that typically represents the address of a user account. For instance, the expression `exists<Coin<USD>>(addr)` will be true if there is a value of type `Coin<USD>` stored at `addr`. As seen in this example, Move uses type generics, and working with generic functions and types is rather idiomatic for Move.

A Move package consists of a set of *modules*. Each module defines a set of Move functions. These functions update the global memory and may emit events. The execution of these functions can abort explicitly because of an abort instruction (failure of an `assert`) or implicitly because of a runtime error such as an out-of-bounds vector read or integer overflows. For instance, the `coin` module provides the foundation for coins on Aptos. As one of the core public APIs defined in it, the function `deposit` is shown in Listing 1:

1. checks whether the coin with type `CoinType` is registered under the recipient's account with the address `account_addr`;
2. retrieves a mutable reference to the corresponding resource `CoinStore` from the account;
3. if the store is not frozen, deposits the input `coin` to the `CoinStore` by calling the `merge` function and emits an `Deposit` event.

If the function executes successfully, the borrowed global resource `CoinStore` in `account_addr` will be updated after the transaction is committed. Otherwise, the transaction will abort without any changes to the global memory.

Listing 1 The deposit function of the coin module.

```
public fun deposit<CoinType>(
  account_addr:address,
  coin:Coin<CoinType>
) acquires CoinStore {
  assert!(
    is_account_registered<CoinType>(account_addr),
    error::not_found(ECOIN_STORE_NOT_PUBLISHED)
  );
  let coin_store = borrow_global_mut<CoinStore<CoinType>>(account_addr);
  assert!(!coin_store.frozen, error::permission_denied(EFROZEN));
  event::emit(
    Deposit<CoinType> { account: account_addr, amount: coin.value }
  );
  merge(&mut coin_store.coin, coin);
}
```

2.2 Move Prover

The Move Prover (MVP for short) [21] is a formal verification tool for smart contracts that are written in the Move language. The Move language is tightly coupled and integrated with MVP because they have been developed and are evolving together. Move features an expressive specification language designed to define the intended behaviors of a Move smart contract. The architecture of MVP is shown in Figure 1. Move code (with the specification) is given as input to the tool chain, which produces two artifacts: an abstract syntax tree (AST) of the specifications, and the generated bytecode. They are merged into a unified object model and input to the Prover Compiler. After a series of bytecode transformations such as reference elimination, specification instrumentation and monomorphization, Boogie [14] IR is generated which is further lowered into the SMT language and subsequently fed to an SMT solver such as Z3 [6] or CVC5 [18]. MVP checks whether the code satisfies the user-given specification for all possible program variable assignments. If not, MVP generates a counterexample, that is an assignment to program variables such that the specification does not hold. The Move Prover takes great care of translating the counter-example back into the Move representation, hiding the intrinsic details of the SMT solver. MVP is fast and reliable [8], and can be used routinely during smart contract development, making the experience of running MVP similar to the experience of running compilers, linters, type checkers, and other development tools.

2.3 Move Specification Language

The Move specification language allows developers to specify the properties of their smart contracts, leveraging MVP to guarantee they behave as specified without adding any runtime cost on-chain. In the specification language, developers can provide pre- and post-conditions for functions, which include conditions over input parameters and global memory. Developers may also provide invariants over data structures as well as the contents of the global memory. The language also supports universal and existential quantification over bounded domains, such as the indices of a vector, as well as effectively unbounded domains, such as memory addresses and integers (e.g., `forall a: address: P(a)`, and `exists i: u64: Q(i)`, for some predicates `P` and `Q`). While quantifiers can render the verification problem undecidable

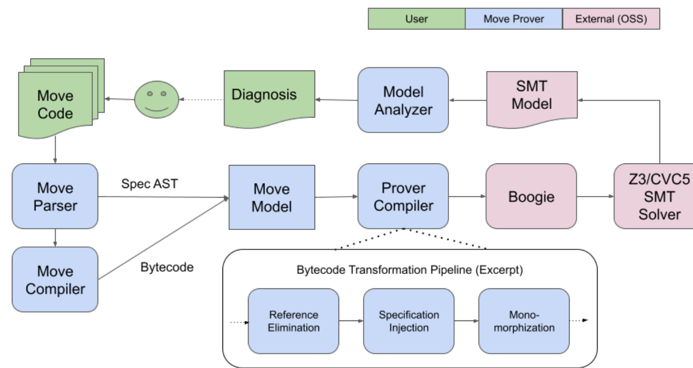

Figure 1 Architecture of the Move Prover.

and lead to timeout or an "unknown" response from SMT solvers, they offer a practical advantage: they allow for a more direct formalization of various properties, enhancing the clarity of specifications.

As an example, the spec block in Listing 2 shows the specification of the `deposit` function described in Section 2.1, which is the mathematical representation of the expected behavior of the function. Two `aborts_if` clauses specify that the function aborts if and only if at least one of the following conditions are satisfied: (1) the `CoinStore` resource for the coin with the type `CoinType` does not exist under the account `account_addr`; (2) the `CoinStore` resource is frozen. The `ensures` clause specifies that the value of the coin stored under `account_addr` is increased by the value of the input `coin` after execution. MVP guarantees that function implementation satisfies this specification for all input values and all coin types. MVP 's formal verification contrasts with testing, where a single test case only covers a specific instance of input and coin type. Moreover, once a specification for MVP is defined, it enables MVP to automatically check the specification thereafter (through CI). This automation significantly reduces the costs associated with repetitive manual audits for every modification of the smart contract.

Listing 2 The spec of the deposit function.

```
spec deposit<CoinType>(
  account_addr: address,
  coin: Coin<CoinType>
) {
  aborts_if !exists<CoinStore<CoinType>>(account_addr);
  aborts_if global<CoinStore<CoinType>>(account_addr).frozen;
  ensures global<CoinStore<CoinType>>(account_addr)).coin.value ==
    old(global<CoinStore<CoinType>>(account_addr)).coin.value +
    coin.value;
}
```

2.4 Aptos Framework

The Aptos Framework defines standard actions performed on-chain. For instance, the `genesis` module defines operations to be executed during genesis such as initializing the core account and core modules on chain. The `block_prologue` function in the `block` module defines the actions to execute before each transaction, updating the current block's metadata and the on-chain performance scores of validators. Beyond system-related actions, the framework establishes standards for coins and staking, math libraries, efficient data structures (e.g.,

smart vectors which adapt depending on their size), and cryptographic algorithms (e.g., ed25519). Given its essential role in the Aptos Network, security and safety of the framework are of utmost importance: bugs can cause network disruptions or lead to significant financial losses. We conducted thorough testing and auditing of the Aptos Framework, however, it is well-known that these measures cannot guarantee the complete absence of bugs [7]. Formal verification, in contrast, can provide rigorous proofs of critical properties. In the next section, we will present how to apply this technique effectively to the Aptos Framework.

3 Formal Verification of the Aptos Framework

Formal verification has received significant attention in the blockchain industry due to the critical importance of smart contract assurance [15, 19]. However, applying formal verification to a smart contract framework is challenging, especially when it encompasses tens of thousands of lines of code and undergoes constant evolution. Moreover, a formally verified smart contract is only as correct as its specifications. Therefore, how to devise a comprehensive set of specifications for large and evolving codebase is the key to maximize the return on investment (ROI) in formal methods.

To address this challenge, we have adopted two approaches to devising specification from distinct directions. First, we adopted a top-down approach, starting from high-level requirements to detailed specifications. Inspired by prior work of the authors [10], we established a traceability framework that enables the tracking of how high-level requirements are tested, audited, and verified. The high-level requirements and the traceability information are thoroughly documented within the Move spec files, including the links between high-level requirements and their respective formal specs. This ensures that all critical safety properties are covered in specifications with provenance. In addition, we pursued a bottom-up approach – systematically deriving specifications from individual functions and modules. This approach allowed us to uncover, verify, and document functional properties of the Aptos Framework. This section explains this combined effort in more detail.

3.1 From Security Requirement to Verification

In collaboration with our audit firm, we identified critical security requirements for each module within the Aptos Framework. These requirements were systematically documented with details covering their definition, criticality, implementation approach, and enforcement methods. The enforcement methods include audit, test, and, where appropriate, formal verification. For each requirement that can be enforced by formal verification, we created the corresponding formal specifications and verified them with MVP.

For example, a high-critical security requirement for the `coin` module is shown in Table 1. It states that the supply of a coin can be changed only by certain operations, such as `burn` and `mint`. This requirement is in place to ensure that coins cannot be created arbitrarily, which could potentially result in significant financial loss. The implementation has been audited manually, and the property has been further specified and verified.

Shown in Listing 3, this requirement is encoded as a post-condition to be applied to all functions in the `coin` module except for the `mint` and `burn` functions.

Listing 3 Post condition to enforce the high-level requirement in Table 1.

```
spec module {
  apply TotalSupplyNoChange<CoinType> to *<CoinType>
    except mint, burn, burn_from;
}
```

Table 1 A high-level requirement of the coin module.

No.	Requirement	Criticality	Implementation	Enforcement
4	The supply of a coin is only affected by burn and mint operations.	High	Only `mint` and `burn` operations on a coin alter the total supply of coins.	Formally verified in `TotalSupplyNoChange`

The definition of the post-condition `TotalSupplyNoChange` is given in Listing 4. The condition explicitly states that the `supply` value remains unchanged when comparing its values before and after execution. It uses two concepts not seen so far: *specification functions* which allow to work on state and appear in `old(..)` expressions, as well as *specification schemas* which allow to group and later inject properties:

Listing 4 Definition of TotalSupplyNoChange.

```
spec fun supply<CoinType>(addr: address): u128 {
  option::spec_borrow(global<CoinType>(addr)).supply
}
spec schema TotalSupplyNoChange<CoinType> {
  coin_type_address: address;
  ensures option::is_some(global<CoinType>(coin_type_address)) ==> supply
      (coin_address) == old(supply(coin_address))
}
```

We have authored the high-level requirements for over 40 modules, and most of the requirements are formally verified (see Appendix A). We also integrated these high-level requirement artifacts into the Aptos Framework reference documentation. Additionally, we've implemented a traceability framework that establishes connections between high-level requirements and their associated formal specifications, facilitating the mapping of formal specifications back to their originating high-level requirements. For example, the high-level requirements of the coin module can be found in the link[3]. For other modules, please see Appendix A.

3.2 Systematic functional specification

By verifying high-level requirements, we ensure that the Aptos Framework satisfies the critical security properties identified during the audit. It is also important to ensure the functionally correct behavior of the Aptos Framework. To ensure functional correctness, we have systematically inferred the specifications from the code and comments of Move functions and modules through a thorough manual process. This involves examining each Move function to identify its abort conditions and post-conditions. Additionally, we meticulously examined every struct to determine its data invariants and, by synthesizing these findings, also established global invariants for each module.

3.2.1 Abort conditions

`aborts_if` specifications cover important classes of properties, such as access control checks, input validation, and state validation. Move functions are normally designed to abort when they are called (1) by an account without permission, (2) with an input argument outside of an expected range, or (3) on an unexpected global state. For example,

[3] `https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-framework/doc/coin.md#high-level-req`

in the Aptos Framework's staking config contract, only the `aptos_framework` account (i.e., `0x1`) can call `update_recurring_lockup_duration_secs`. Also, the input parameter `new_recurring_lockup_duration_secs` should be non-zero. It should be called only when the resource `StakingConfig` is published under the `aptos_framework` address. These expected behaviors are captured by the specification in Listing 5.

Listing 5 Specification of update_recurring_lockup_duration_secs.

```
spec update_recurring_lockup_duration_secs(
    aptos_framework: &signer,
    new_recurring_lockup_duration_secs: u64
){
    aborts_if signer::address_of(aptos_framework) != @aptos_framework;
    aborts_if new_recurring_lockup_duration_secs == 0;
    aborts_if !exists<StakingConfig>(@aptos_framework);
    ...
}
```

Given this `aborts_if` specification, MVP verifies two things. First, it verifies that the function indeed aborts when any one of the conditions holds. Second, MVP verifies that the function does not abort on any other condition. This verification is important because it allows developers to understand the complete set of conditions under which the function can abort, thus the specifications also serve as precise documentation. Notice that abort condition verification works very smoothly with MVP in practice, as finding a particular program point that aborts is a simpler problem for the solver than general post-conditions. In virtually any case we have encountered, a missed abort is quickly identified and pointed to in the Move source.

For certain functions, it is critical that they do not abort. For instance, the `block_prologue` function must never abort since it is executed with each block, and a malfunction can bring the entire network down. The block prologue complex execution involves 96 Move functions in 22 different Move modules. We formally specified all these modules and proved that the block prologue execution would never fail (or abort) in an unexpected condition (see Appendix A). The top-level specification of this function can be found in Listing 6, with some schemas included which contain further details. The clause "`aborts_if false`" means that this function should never abort, which can be proven if the `requires` conditions over the input and the global state hold. For instance, the third `requires` condition says the proposer (one of the input arguments) of creating the block must be either a reserved address (`@vm_reserved`) or an active validator, which needs to be checked by retrieving a global resource in the `stake` module. It is worth noting that this function is directly called by the VM, so conditions in the specification were manually audited at the call site.

Listing 6 Specification of block_prologue.

```
spec block_prologue {
    requires chain_status::is_operating();
    requires system_addresses::is_vm(vm);
    requires proposer == @vm_reserved
             || stake::spec_is_current_epoch_validator(proposer);
    requires timestamp >= reconfiguration::last_reconfiguration_time();
    requires (proposer == @vm_reserved)
             ==> (timestamp::spec_now_microseconds() == timestamp);
    requires (proposer != @vm_reserved)
              ==> (timestamp::spec_now_microseconds() < timestamp);
    requires exists<stake::ValidatorFees>(@aptos_framework);
    requires exists<CoinInfo<AptosCoin>>(@aptos_framework);
```

```
    include transaction_fee::
        RequiresCollectedFeesPerValueLeqBlockAptosSupply;
    include staking_config::StakingRewardsConfigRequirement;
    aborts_if false; // can never abort
}
```

3.2.2 Struct invariants

Struct invariants define the properties that the data within a struct must consistently satisfy. When specifying the Aptos Framework, we observed many cases where the invariants of a struct were implicitly present. These invariants were often documented within code comments or manifested as assertion statements, and related functions were developed while observing these implicit invariants. We have explicitly specified the invariants and verified them using MVP. For example, the struct `GasCurve` (shown in Listing 7) represents a gas curve to be used to adjust the global storage gas. It is an Eulerian approximation of an exponential curve. The fields `min_gas` and `max_gas` are the minimum and maximum gas charges respectively, and `points` is a vector of (x, y) pairs that represent the basis points of the curve where the `x`-coordinate is the utilization ratio in the curve and `y`-coordinate is the utilization multiplier in the curve.

Listing 7 Definition of GasCurve and Point.

```
struct GasCurve has copy, drop, store {
  min_gas: u64,
  max_gas: u64,
  points: vector<Point>
}
struct Point has copy, drop, store {
  x: u64,
  y: u64
}
```

For each time period, the storage gas is recalculated by interpolation into the curve that `GasCurve` defines. An implicit invariant of `GasCurve` was documented within code comments to ensure correct linear interpolation. It says that every instance of `GasCurve` is well-formed, representing a properly structured curve. Otherwise, interpolation becomes impossible, leading to an abortion of the process. We formally specified the invariant to ensure it is enforced consistently in all places. Listing 8 shows the specification of the invariant. For each point instance, the (x,y) pair must not exceed the basis point denomination. For each gas curve instance, 1) the minimum gas charge does not exceed the maximum gas charge; 2) the maximum gas charge is capped by `MAX_U64` scaled down by the basis point denomination; and 3) the gas curve is a monotonically increasing function. MVP ensures that those invariants hold everywhere in the code, that is, whenever of a value of the according types is constructed or modified. It is worth mentioning that the storage gas recalculation is part of the `block_prologue` execution path; thus, the data invariant of `GasCurve` plays an important role in the `block_prologue` verification.

Listing 8 Invariants of GasCurve and Point.

```
spec GasCurve {
  invariant min_gas <= max_gas;
  invariant (len(points) > 0 ==> points[0].x > 0);
  invariant forall i in 0..len(points) - 1:
    (points[i].x < points[i + 1].x && points[i].y <= points[i + 1].y);
  invariant max_gas <= MAX_U64 / BASIS_POINT_DENOMINATION;
}
```

```
spec Point {
  invariant x <= BASIS_POINT_DENOMINATION;
  invariant y <= BASIS_POINT_DENOMINATION;
}
```

3.2.3 Global invariants

Global invariants define the properties that the global state must consistently satisfy. Global invariants appear as members of the module specification. They are expressed as conditions over the global state that consist of Move resources published in the global memory space. Global invariants are important properties because they specify and ensure the correctness of the entire global state. Several global invariants have been inferred from some core modules of the Aptos Framework. For instance, the `stake` module defines Aptos' staking mechanism. Listing 9 shows the struct definitions related to validators. The resource `ValidatorSet` contains the configuration information for the active validators of the Aptos Network. In the struct `ValidatorConfig`, the field `validator_index` field denotes the index within the active validator set. Its value is updated following changes to the active validator set.

Listing 9 Struct definitions related to validators.

```
struct ValidatorSet has key {
  active_validators: vector<ValidatorInfo>,
  // other fields...
}
struct ValidatorInfo has copy, store, drop {
  addr: address,
  voting_power: u64,
  config: ValidatorConfig,
}
struct ValidatorConfig has key, copy, store, drop {
  validator_index: u64,
  // other fields...
}
```

In the function `update_stake_pool` (shown in Listing 10), the field `validator_index` is used to locate the corresponding validator's performance (i.e., the number of successful proposals) data entry in `ValidatorPerformance`. The function will abort if `validator_index` is equal to or greater than the length of `validators` in `ValidatorPerformance`.

Listing 10 Definition of the function update_stake_pool.

```
fun update_stake_pool(
  validator_perf: &ValidatorPerformance,
  pool_address: address,
  staking_config: &StakingConfig
) {
  let validator_config = borrow_global<ValidatorConfig>(pool_address);
  let cur_validator_perf = vector::borrow(
    &validator_perf.validators,
    validator_config.validator_index
  );
  // ...
}
```

To prove that `update_stake_pool` never aborts unexpectedly, it is necessary to establish the fact that `validator_index` never holds a value that is out-of-bounds for `validators` in `ValidatorPerformance`. Listing 11 shows the formal specification of the global invariant, which denotes the correct relation between the two resources `ValidatorSet` and `ValidatorPerformance` in the global memory. The invariant says that if the resource

ValidatorSet exists, all values of the validator_index fields in ValidatorSet must be smaller than the length of validators in ValidatorPerformance. Notice that this property cannot be expressed by a data invariant since those must not depend on global memory, but here, we indirectly index global memory by an address found in a ValidatorSet.

Listing 11 Global invariant on validators.

```
spec module {
  invariant exists<ValidatorSet>(@aptos_framework) ==>
    validator_set_is_valid();
  fun validator_set_is_valid(): bool {
    let set = global<ValidatorSet>(@aptos_framework);
    forall i in 0..len(set.active_validators):
      global<ValidatorConfig>(validators[i].addr).validator_index <
        len(global<ValidatorPerformance>(@aptos_framework).validators)
  }
}
```

4 Discussion

In this section, we discuss the benefits of formal verification for enhancing the security of the Aptos Framework, along with insights gained from this endeavor. Formal verification not only establishes proofs for key properties of the Aptos Framework but also aids in identifying bugs and issues. The process of writing formal specifications demands thorough code review, while the analysis of counterexamples unveils nuanced program behaviors, enabling the detection of certain bugs. Throughout the formal specification and verification process, numerous issues were identified, including the aptos_governance::store_signer_cap example shown in Listing 12.

Listing 12 Incorrect implementation of store_signer_cap.

```
public fun store_signer_cap(
  aptos_framework: &signer,
  signer_address: address,
  signer_cap: SignerCapability,
) acquires GovernanceResponsbility {
  system_addresses::assert_framework_reserved_address(
    address_of(aptos_framework)
  );
  if (!exists<GovernanceResponsbility>(@aptos_framework)) {
    move_to(aptos_framework, GovernanceResponsbility {
      signer_caps: simple_map::create<address,SignerCapability>()
    });
  };
  let signer_caps = &mut borrow_global_mut<GovernanceResponsbility>(
    @aptos_framework
  ).signer_caps;
  simple_map::add(signer_caps, signer_address, signer_cap);
}
```

Listing 12 shows the incorrect version of the function prior to our correction, caused by a subtle difference between the signer argument aptos_framework and the address constant @aptos_framework. This function misbehaves when address_of(aptos_framework) differs from @aptos_framework. While the address constant @aptos_framework is set to be 0 x1, the argument aptos_framework can represent any reserved address from 0x1 to 0xa because the function checks if aptos_framework corresponds to one of these reserved addresses. Consequently, the resource GovernanceResponsibility might be established under address_of(aptos_framework), which might not align with @aptos_framework. This

discrepancy raises the possibility that the resource `GovernanceResponsibility` may not be present under `@aptos_framework`, leading to the potential abort in `borrow_global_mut<GovernanceResponsibility>(@aptos_framework)`.

Listing 13 Specification of store_signer_cap.

```
spec store_signer_cap(
  aptos_framework: &signer,
  signer_address: address,
  signer_cap: SignerCapability,
) {
  aborts_if !system_addresses::is_aptos_framework_address(
    address_of(aptos_framework)
  );
  aborts_if !system_addresses::is_framework_reserved_address(
    signer_address
  );
  let signer_caps = global<GovernanceResponsbility>(
    @aptos_framework
  ).signer_caps;
  aborts_if exists<GovernanceResponsbility>(@aptos_framework) &&
    simple_map::spec_contains_key(signer_caps, signer_address);
  ensures exists<GovernanceResponsbility>(@aptos_framework);
}
```

Listing 13 formally specifies the intended behavior of the function `store_signer_cap`, which could not be verified against its incorrect version of the function. To resolve this, we amended the code to ensure that `aptos_framework` matches the address `@aptos_framework`, and that `signer_address` is indeed a reserved address.

Moreover, we have identified and defined various invariants within the modules, significantly enhancing our understanding of their behaviors and their security implications. This deeper insight has subsequently guided the refactoring and improvement of several modules. For instance, the `on_new_epoch` function in the stake module initially had a complex and lengthy while loop. In specifying the function, we divided this while loop into two separate loops. This division not only simplified the writing of loop invariants but also enhanced their readability and understandability. Furthermore, in the process of specifying the `calculate_reward_amount` function, we identified multiple issues, including overflow, rounding errors, and division by zero. To address these concerns, we refactored the code to use higher precision `u128` instead of `u64` in the intermediate steps to avoid overflow, performed division at the end to minimize rounding errors, and ensured the denominator was non-zero before division to prevent division-by-zero errors. Listing 14 presents the `calculate_reward_amount` post-refactoring.

Listing 14 the calculate_reward_amount function of the stake module.

```
fun calculate_rewards_amount(
  stake_amount: u64,
  num_successful_proposals: u64,
  num_total_proposals: u64,
  rewards_rate: u64,
  rewards_rate_denominator: u64
): u64 {
  let rewards_numerator = (stake_amount as u128) *
    (rewards_rate as u128) * (num_successful_proposals as u128);
  let rewards_denominator = (rewards_rate_denominator as u128) *
    (num_total_proposals as u128);
  if (rewards_denominator > 0) {
    ((rewards_numerator / rewards_denominator) as u64)
  } else {
    0
  }
}
```

The specifications for the Aptos Framework developed in this work are also an important component of the automatically generated reference documentation for the Aptos Framework, offering a comprehensive and precise description of the expected behavior of each function and module. This detailed documentation is vital to ongoing maintenance and quality assurance efforts. For example, the `requires` conditions specified for the `block_prologue` function, as illustrated in Listing 6, serve as assumptions that cannot be verified directly because the function is invoked by the VM, and VM verification falls outside the scope of MVP. Nonetheless, these conditions clearly document all the assumptions for the VM regarding invoking `block_prologue`, thus enabling a more streamlined and time-effective manual audit process.

Here are some key lessons learned: adopting a top-down approach proved beneficial, as it allowed us to grasp and establish high-level security requirements comprehensively, ensuring no critical security aspect was overlooked. These high-level requirements serve as an effective communication bridge between security experts and developers, proving invaluable for future code maintenance. However, verifying high-level requirements alone can be challenging without the support of local and global specifications, such as those provided through `aborts_if` specs and data/global invariants. Therefore, a bottom-up approach becomes essential for systematically developing these specifications. Moreover, integrating MVP into the Continuous Integration (CI) testing process significantly reduces the necessity for frequent audits with each contract modification. Consequently, formal verification has effectively decreased both the time and cost associated with auditing.

We encountered several challenges in writing local and global specifications, particularly due to the difficulties in writing loop invariants and timeouts with MVP. To overcome these issues, we employed several abstraction methods: (1) modeling non-linear functions as non-interpreted functions to facilitate verification of their callers; (2) applying loop unrolling techniques for complex or nested loops to enable bounded checking; and (3) for functions operating over large data domains like `u128` (e.g., `math128`), we performed verification within a smaller data domain such as `u8`. This strategy allowed for effective verification of non-linear functions like `max`, `min`, `mul_div`, `clamp`, `pow`, `floor_log2`, `sqrt`, and `ceil_div` within the domain of `u8`. However, there is more work to do specifically regarding loops in Move programs, which we hope to avoid as much as possible and instead be replaced by higher-order functions like `foreach`, `map`, `fold` and so on, which can then be specially treated by MVP.

The verification artifacts for this work can be accessed online. Appendix A offers a collection of annotated links to these artifacts. Also, Appendix B details the verification results (e.g., the number of verification conditions and the execution time) and outlines the steps for their reproduction.

5 Related Work

Many approaches have been applied to the verification of smart contracts; see e.g. the surveys [15, 19]. [19] refers to at least two dozen systems for smart contract verification. It distinguishes between *contract* and *program* level approaches. Our approach has aspects of both: we address program level properties via pre/post conditions, and contract ("blockchain state") level properties via global invariants. Among the existing approaches, the Move ecosystem is the first one where contract programming and specification language are fully integrated, and the language is designed from the first principles influenced by verification.

Methodologically, Move and MVP are thereby closer to systems like Dafny [13], or the older Spec# system [3], where instead of adding a specification approach posterior to an existing language, it is part of it from the beginning.

In contrast to other approaches that only focus on specific vulnerability patterns [5, 16, 17, 20], MVP offers a universal specification language. We support universal quantification over arbitrary memory content as well as global invariants. For comparison, the SMT Checker for Solidity [9, 11, 12] does not support quantifiers, because it interprets programming language constructs (requires and assert statements) as specifications and has no dedicated specification language. While in Solidity, one can simulate aspects of global invariants using modifiers by attaching pre/post conditions, this is not the same as our invariants, which are guaranteed to hold independent of whether a user may or (accidentally) may not attach a modifier. Moreover, our invariants are optimized to be evaluated only when necessary.

The Certora verifier [4] is a formal verification tool for Solidity smart contracts which has expressiveness comparable to that of MVP. However, because Move bytecode is fully typed and of a higher abstraction level than EVM bytecode, the verification task of Certora becomes substantially more challenging compared to MVP. This complexity could potentially render MVP more user-friendly and accessible for application.

6 Conclusion

To ensure the highest standards of quality and security within the Aptos Network, we have rigorously applied formal verification techniques to the Aptos Framework. We thoroughly documented high-level security requirements. Subsequently, we specified each aspect of the Aptos Framework, function-by-function, until most of the high-level security requirements and functional properties were eventually formally verified. During this process, we also found and fixed bugs and usability issues within MVP, thus benefiting all Aptos developers. This combined work gives the Aptos Framework a high level of quality assurance and, to the best of our knowledge, represents one of the first large-scale smart contract verification. Looking ahead, Aptos Labs plans to maintain and evolve the verification work as well as improve MVP itself. This effort aims to ensure the tool stays available for developers and auditors in the Aptos ecosystem, thereby enhancing software quality in the smart contract domain.

References

1 Aptos. The Aptos Blockchain. `https://aptos.dev/aptos-white-paper`, 2022.

2 Aptos Labs, MoveBit, and OtterSec. Securing the Aptos Framework through formal verification. `https://medium.com/aptoslabs/securing-the-aptos-framework-through-formal-verification-14124d1ed660`, 2024.

3 Mike Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. *The Spec# Programming System: Challenges and Directions*, pages 144–152. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. `doi:10.1007/978-3-540-69149-5_16`.

4 Certora. Certora Prover Documentation. `https://docs.certora.com/en/latest/index.html`, 2022.

5 ConsenSys. Mythril Classic: Security analysis tool for Ethereum smart contracts. URL: `https://github.com/skylightcyber/mythril-classic`.

6 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

7. E. W. Dijkstra. *On the Reliability of Programs*, pages 359–370. Association for Computing Machinery, New York, NY, USA, 1 edition, 2022. doi:10.1145/3544585.3544608.
8. David L. Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Jingyi Emma Zhong. Fast and reliable formal verification of smart contracts with the move prover (extended version). *CoRR*, abs/2110.08362, 2021. arXiv:2110.08362.
9. Ethereum Foundation. Solidity documentation, 2018. URL: http://solidity.readthedocs.io.
10. Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Víctor A. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Softw. Test. Verification Reliab.*, 21(1):55–71, 2011. doi:10.1002/STVR.427.
11. Ákos Hajdu and Dejan Jovanovic. solc-verify: A modular verifier for solidity smart contracts. *CoRR*, abs/1907.04262, 2019.
12. Ákos Hajdu and Dejan Jovanovic. SMT-Friendly Formalization of the Solidity Memory Model. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 224–250. Springer, 2020.
13. K. M. Leino. Accessible software verification with dafny. *IEEE Software*, 34(06):94–97, November 2017. doi:10.1109/MS.2017.4121212.
14. K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, pages 312–327, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
15. Jing Liu and Zhentian Liu. A survey on security verification of blockchain smart contracts. *IEEE Access*, 7:77894–77904, 2019.
16. Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *ACM Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.
17. Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *ACSAC*, pages 653–663. ACM, 2018.
18. The CVC Team. CVC5. URL: https://github.com/cvc5/cvc5.
19. Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *CoRR*, abs/2008.02712, 2020. arXiv:2008.02712.
20. Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *ACM Conference on Computer and Communications Security*, pages 67–82. ACM, 2018.
21. Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L. Dill. The Move Prover. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 137–150. Springer International Publishing, 2020.

A Verification Artifacts

This section overviews the artifacts of this verification work. The Aptos Framework consists of three Move packages such as

- `move-stdlib`: the common standard library of vanilla Move,
- `aptos-stdlib`: the Aptos-specific standard library,
- `aptos-framework`: the Aptos' standard modules for coin, staking, voting, and other operations

We've specified and verified all three Move packages. The Move modules and specs can be found via the following permanent links:

- `move-stdlib` modules and specs:
  - (modules) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/move-stdlib/doc/overview.md#module-index

  - (specs) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/move-stdlib/doc/overview.md#specification-index
- aptos-stdlib modules and specs:
  - (modules) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-stdlib/doc/overview.md#module-index
  - (specs) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-stdlib/doc/overview.md#specification-index
- aptos-framework modules and specs:
  - (modules) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-framework/doc/overview.md#module-index
  - (specs) https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-framework/doc/overview.md#specification-index

Moreover, the high-level security requirement document is an important artifact of this verification work. The high-level security requirements are presented in the markdown table formats within the Aptos Framework's reference documentation, including hyperlinks to the relevant formal specifications, facilitating requirement traceability. All the high-level security requirements of aptos-framework can be accessed through this index: https://github.com/aptos-labs/aptos-core/blob/fmbc-24/aptos-move/framework/aptos-framework/doc/overview.md#high-level-security-requirement-index. Please note that if the browser does not automatically navigate to the item linked from the index pages above, refreshing your browser should resolve the issue and direct you to the correct item.

The Aptos Framework is open-source and accessible on GitHub. Each module in move-stdlib and its spec is contained in a single Move source file .move, while aptos-stdlib and aptos-framework define modules in .move and their specs in .spec.move separately. The source code of Move modules and specs can be found at the following link:

- move-stdlib: https://github.com/aptos-labs/aptos-core/tree/fmbc-24/aptos-move/framework/move-stdlib/sources
- aptos-stdlib: https://github.com/aptos-labs/aptos-core/tree/fmbc-24/aptos-move/framework/aptos-stdlib/sources
- aptos-framework: https://github.com/aptos-labs/aptos-core/tree/fmbc-24/aptos-move/framework/aptos-framework/sources

B Reproducing the Verification Result

This section explains how to reproduce the verification result. The steps are summarized as follows:

1. Install Aptos CLI. The Move Prover (MVP) is integrated in the Aptos CLI[4], a command line tool for developing, debugging, deploying and operating on the Aptos Network. To install the Aptos CLI, please refer to https://aptos.dev/tools/aptos-cli/install-cli/.
2. Clone the aptos-core repository. aptos-core contains the core components of the Aptos Network including the Aptos Framework and their specs. Please use the following command to download the snapshot of the repository made for this paper: `git clone --branch fmbc-24 git@github.com:aptos-labs/aptos-core.git`

[4] https://aptos.dev/tools/aptos-cli/

3. Install the dependencies of MVP. MVP requires the backend verification tools such as Z3 and Boogie. To install all the dependencies that MVP needs, please run the command `./script/dev_setup -ytp` in the `aptos-core` directory, and execute the environment command in `.profile` to properly set the environment variables such as `Z3_EXE` and `BOOGIE_EXE`.
4. Run MVP. To prove the `aptos-framework` package, please go to the `aptos-move/framework/aptos-framework` directory and run `aptos move prove` (`aptos-move/framework/move-stdlib` for `move-stdlib` and `aptos-move/framework/aptos-stdlib` for `aptos-stdlib`).

Listing 15 shows the verification result that has been performed on a Apple M1 Max machine with 64 GB of memory.

Listing 15 The verification result.

```
# The verification result for move-stdlib
[INFO] preparing module 0x1::BCS
...
[INFO] preparing module 0x1::string
[INFO] transforming bytecode
[INFO] generating verification conditions
[INFO] 138 verification conditions
[INFO] running solver
[INFO] 0.049s build, 0.023s trafo, 0.013s gen, 2.027s verify, total 2.113
    s
Success

# The verification result for aptos-stdlib
[INFO] preparing module 0x1::bls12381_algebra
...
[INFO] preparing module 0x1::smart_vector
[INFO] transforming bytecode
[INFO] generating verification conditions
[INFO] 338 verification conditions
[INFO] running solver
[INFO] 0.204s build, 0.124s trafo, 0.045s gen, 18.347s verify, total
    18.721s
Success

# The verification result for aptos-framework
[INFO] preparing module 0x1::system_addresses
...
[INFO] preparing module 0x1::staking_proxy
[INFO] transforming bytecode
[INFO] generating verification conditions
[INFO] 525 verification conditions
[INFO] running solver
[INFO] 0.735s build, 0.636s trafo, 0.193s gen, 84.024s verify, total
    85.589s
Success
```

Notice the number of verification conditions prompted for each of those commands corresponds to one function (if generic, one instantiation) and all its pre/post conditions, injected invariants, and properties inlined in the code (e.g. loop invariants).