End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract

Daejun Park[1], Yi Zhang[1,2], and Grigore Rosu[1,2]

[1] Runtime Verification, Inc.
daejun.park@runtimeverification.com
[2] University of Illinois at Urbana-Champaign
{yzhng173,grosu}@illinois.edu

Abstract. We report our experience in the formal verification of the deposit smart contract, whose correctness is critical for the security of Ethereum 2.0, a new Proof-of-Stake protocol for the Ethereum blockchain. The deposit contract implements an incremental Merkle tree algorithm whose correctness is highly nontrivial, and had not been proved before. We have verified the correctness of the compiled bytecode of the deposit contract to avoid the need to trust the underlying compiler. We found several critical issues of the deposit contract during the verification process, some of which were due to subtle hidden bugs of the compiler.

1 Introduction

The deposit smart contract [14] is a gateway to join Ethereum 2.0 [15] that is a new sharded Proof-of-Stake (PoS) protocol which at its early stage, lives in parallel with the existing Proof-of-Work (PoW) chain, called Ethereum 1.x chain. Validators drive the entire PoS chain, called Beacon chain, of Ethereum 2.0. To be a validator, one needs to deposit a certain amount of Ether, as a "stake", by sending a transaction (over the Ethereum 1.x network) to the deposit contract. The deposit contract records the history of deposits, and locks all the deposits in the Ethereum 1.x chain, which can be later claimed at the Beacon chain of Ethereum 2.0.[3] Note that the deposit contract is a one-way function; one can move her funds from Ethereum 1.x to Ethereum 2.0, but not vice versa.

The deposit contract, written in Vyper [19], employs the Merkle tree [30] data structure to efficiently store the deposit history, where the tree is *dynamically* updated (i.e., leaf nodes are incrementally added in order from left to right) whenever a new deposit is received. The Merkle tree employed in this contract is very large: it has height 32, so it can store up to 2^{32} deposits. Since the size of the Merkle tree is huge, it is not practical to reconstruct the whole tree every time a new deposit is received.

To reduce both time and space complexity, thus saving the gas[4] cost significantly, the contract implements an *incremental Merkle tree algorithm* [6]. The

[3] This deposit process will change at a later stage.

[4] In Ethereum, gas refers to the fee to execute a transaction or a smart contract on the blockchain. The amount of gas fee depends on the size of the payloads.

incremental algorithm enjoys $O(h)$ time and space complexity to reconstruct (more precisely, compute the root of) a Merkle tree of height h, while a naive algorithm would require $O(2^h)$ time or space complexity. The efficient incremental algorithm, however, leads to the deposit contract implementation being unintuitive, and makes it non-trivial to ensure its correctness. The correctness of the deposit contract, however, is critical for the security of Ethereum 2.0, since it is a gateway for becoming a validator. Considering the utmost importance of the deposit contract for the Ethereum blockchain, formal verification is demanded to ultimately guarantee its correctness.

In this paper, we present our formal verification of the deposit contract.[5] The scope of verification is to ensure the correctness of the contract bytecode within a single transaction, without considering transaction-level or off-chain behaviors. We take the compiled bytecode as the verification target to avoid the need to trust the compiler.[6]

We adopt a refinement-based verification approach. Specifically, our verification effort consists of the following two tasks:

- Verify that the incremental Merkle tree algorithm implemented in the deposit contract is *correct* w.r.t. the original full-construction algorithm.
- Verify that the compiled bytecode is *correctly generated* from the source code of the deposit contract.

Intuitively, the first task amounts to ensuring the correctness of the contract source code, while the second task amounts to ensuring the compiled bytecode being a sound refinement of the source code (i.e., translation validation of the compiler). This refinement-based approach allows us to avoid reasoning about the complex algorithmic details, especially specifying and verifying loop invariants, directly at the bytecode level. This separation of concerns helped us to save a significant amount of verification effort. See Section 2 for more details.

Challenges. Formally verifying the deposit contract was challenging. First, the algorithm employed in the contract is sophisticated and its correctness is not straightforward to prove. Indeed, we found a critical bug in the algorithm implementation which had been not detected by existing tests. (Section 5.1)

Second, we had to take the compiled bytecode as the verification target, which is much larger (consisting of ∼3,000 instructions) and more complex than the source code. The source-code-level verification was not accepted by the customer for the end-to-end correctness guarantee, especially considering the fact that the compiler is not mature enough [11]. Indeed, we found several new critical bugs in the compiler during the formal verification process. (Section 5.2)

Third, we had to consider not only the functional correctness, but also security properties of the contract. That is, we had to identify the behaviors of the

[5] This was done as part of a contract funded by the Ethereum Foundation [16].

[6] Indeed, we found several new critical bugs [41,42,43,44] of the Vyper compiler in the process of formal verification. See Section 5 for more details.

contract in exceptional cases, and check if they are exploitable. We found a bug of the contract in case that it receives invalid inputs. (Section 5.3)

Finally, we had to take into account potential future changes in the Ethereum blockchain system (called hard-forks). That is, we had to verify that the compiled bytecode will work not only in the current system, but also in any future version of the system that employs a different gas fee schedule. Considering such potential changes of the system required us to generalize the semantics of bytecode execution. We also found a bug regarding that. (Section 5.4)

2 Our Refinement-Based Verification Approach

We illustrate our refinement-based formal verification approach used in the deposit contract verification. We present our approach using the K framework and its verification infrastructure [51,54,45], but it can be applied to other program verification frameworks.

Let us consider a `sum` program that computes the summation from 1 to n:

```
int sum(int n) { int s = 0; int i = 1;
                 while(i <= n) { s = s + i; i = i + 1; } return s; }
```

Given this program, we first manually write an abstract model of the program in the K framework [51]. Such a K model is essentially a state transition system of the program, and can be written as follows:

```
rule: sum(n) ⇒ loop(s: 0, i: 1, n: n)
rule: loop(s: s, i: i, n: n) ⇒ loop(s: s+i, i: i+1, n: n) when i ≤ n
rule: loop(s: s, i: i, n: n) ⇒ return(s) when i > n
```

These transition rules correspond to the initialization, the `while` loop, and the return statement, respectively. The indexed tuple `(s:` s`, i:` i`, n:` n`)` represents the state of the program variables `s`, `i`, and `n`.[7]

Then, given the abstract model, we specify the functional correctness property in reachability logic [53], as follows:

```
claim: sum(n) ⇒ return(n(n+1)/2) when n > 0
```

This reachability claim says that `sum(`n`)` will eventually return $\frac{n(n+1)}{2}$ in all possible execution paths, if n is positive. We verify this specification using the K reachability logic theorem prover [54], which requires us only to provide the following loop invariant:[8]

```
invariant: loop(s: i(i−1)/2, i: i, n: n) ⇒ return(n(n+1)/2) when 0 < i ≤ n+1
```

[7] Note that this abstract model can be also automatically derived by instantiating the language semantics with the particular program, if a formal semantics of the language is available (in the K framework).

[8] The loop invariants in reachability logic mentioned here look different from those in Hoare logic. See the comparison between the two logic proof systems in [54, Section 4]. These loop invariants can be also seen as transition invariants [47].

Once we prove the desired property of the abstract model, we manually refine the model to a bytecode specification, by translating each transition rule of the abstract model into a reachability claim at the bytecode level, as follows:

$$\texttt{claim: evm(pc: pc}_{\texttt{begin}}\texttt{, calldata: \#bytes(32, } n\texttt{), stack: [], } \cdots\texttt{)}$$
$$\Rightarrow \texttt{evm(pc: pc}_{\texttt{loophead}}\texttt{, stack: [0, 1, } n\texttt{], } \cdots\texttt{)}$$
$$\texttt{claim: evm(pc: pc}_{\texttt{loophead}}\texttt{, stack: [}s\texttt{, } i\texttt{, } n\texttt{], } \cdots\texttt{)}$$
$$\Rightarrow \texttt{evm(pc: pc}_{\texttt{loophead}}\texttt{, stack: [}s+i\texttt{, } i+1\texttt{, } n\texttt{], } \cdots\texttt{) when } i \leq n$$
$$\texttt{claim: evm(pc: pc}_{\texttt{loophead}}\texttt{, stack: [}s\texttt{, } i\texttt{, } n\texttt{], } \cdots\texttt{)}$$
$$\Rightarrow \texttt{evm(pc: pc}_{\texttt{end}}\texttt{, stack: [], output: \#bytes(32, } s\texttt{), } \cdots\texttt{) when } i > n$$

Here, the indexed tuple `evm(pc:_, calldata:_, stack:_, output:_)` represents (part of) the Ethereum Virtual Machine (EVM) state, and `#bytes(`N`,`V`)` denotes a sequence of N bytes of the two's complement representation of V.

We verify this bytecode specification against the compiled bytecode using the same K reachability theorem prover [54,45]. Note that no loop invariant is needed in this bytecode verification, since each reachability claim involves only a bounded number of execution steps—specifically, the second claim involves only a single iteration of the loop.

Then, we manually prove the soundness of the refinement, which can be stated as follows: *for any EVM states* σ_1 *and* σ_2*, if* $\sigma_1 \Rightarrow \sigma_2$*, then* $\alpha(\sigma_1) \Rightarrow \alpha(\sigma_2)$, where the abstraction function α is defined as follows:

$$\alpha(\texttt{evm(pc: pc}_{\texttt{begin}}\texttt{, calldata: \#bytes(32, } n\texttt{), stack: [], } \cdots\texttt{)}) = \texttt{sum(}n\texttt{)}$$
$$\alpha(\texttt{evm(pc: pc}_{\texttt{loophead}}\texttt{, stack: [}s\texttt{, } i\texttt{, } n\texttt{], } \cdots\texttt{)}) = \texttt{loop(s: } s\texttt{, i: } i\texttt{, n: } n\texttt{)}$$
$$\alpha(\texttt{evm(pc: pc}_{\texttt{end}}\texttt{, stack: [], output: \#bytes(32, } s\texttt{), } \cdots\texttt{)}) = \texttt{return(}s\texttt{)}$$

Putting all the results together, we finally conclude that the compiled bytecode will return `#bytes(32,`$\frac{n(n+1)}{2}$`)`.

Note that the abstract model and the compiler are *not* in the trust base, thanks to the refinement, while the K reachability logic theorem prover [54,45] and the formal semantics of EVM [24] are.

3 Correctness of the Incremental Merkle Tree Algorithm

In this section, we briefly describe the incremental Merkle tree algorithm of the deposit contract, and formulate its correctness. Both the formalization of the algorithm and the formal proof of the correctness are presented in Appendix A.

A Merkle tree [30] is a perfect binary tree [34] where leaf nodes store the hash of data, and non-leaf nodes store the hash of their children. A *partial Merkle tree up-to* m is a Merkle tree whose first (leftmost) m leaves are filled with data hashes and the other leaves are empty and filled with zeros. The incremental Merkle tree algorithm takes as input a partial Merkle tree up-to m and a new data hash, and inserts the new data hash into the $(m+1)^{\text{th}}$ leaf, resulting in a partial Merkle tree up-to $m+1$.

Figure 1 illustrates the algorithm, showing how the given partial Merkle tree up-to 3 (shown in the left) is updated to the resulting partial Merkle tree up-to

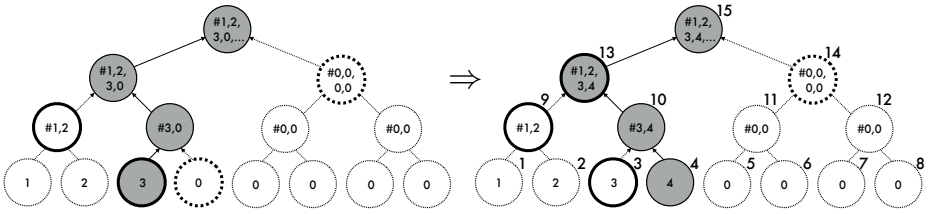


Fig. 1. Illustration of the incremental Merkle tree algorithm. The left tree is updated to the right tree by inserting a new data hash in the fourth leaf (node 4). Only the path from the new leaf to the root (i.e., the gray nodes) are computed by the algorithm (hence linear-time). The bold-lined (and bold-dotted-lined) nodes denote the `branch` (and `zero_hashes`) array, respectively, which are only nodes that the algorithm maintains (hence linear-space). The `#` symbol denotes the hash value, e.g., "`#1,2`" (in node 9) denotes "hash(1,2)", and "`#1,2,3,4`" (in node 13) denotes "hash(hash(1,2),hash(3,4))". Node numbers are labeled in the upper-right corner of each node.

4 (in the right) when a new data hash is inserted into the 4^{th} leaf node. Here are a few key observations of the algorithm.

1. The only difference between the two Merkle trees is the path from the new leaf node (i.e., node 4) to the root. All the other nodes are identical between the two trees.
2. The path can be computed by using only the left (i.e., node 3 and node 9) or right (i.e., node 14) sibling of each node in the path. All the other nodes are *not* needed for the path computation.
3. All the left siblings (i.e., node 3 and node 9) of the path are "finalized" in that they will never be updated in any subsequent execution of the algorithm. All the leaves that are a descendant of the finalized node are non-empty.
4. All the right siblings (i.e., node 14) are zero-hashes, that is, 0 for leaf nodes (at level 0), "hash(0,0)" for nodes at level 1, "hash(hash(0,0),hash(0,0))" for nodes at level 2, and so on. These zero-hashes are constant.

Now we describe the algorithm. To represent a Merkle tree of height h, the algorithm maintains only two arrays of length h, called `branch` and `zero_hashes` respectively, that store the left and right siblings of a path from a new leaf node to the root. When inserting a new data hash, the algorithm computes the path from the new leaf node to the root. Each node of the path can be computed in constant time, by retrieving only its left or right sibling from the `branch` or `zero_hashes` array. After the path computation, the `branch` array is updated to contain all the left siblings of a next new path that will be computed in the next run of the algorithm. Here the `branch` array update is done in constant time, since only a single element of the array needs to be updated, and the element has already been computed as part of the path computation.[9] Note that the `zero_hashes` array is computed once at the very beginning when all the leaves are empty, and never be updated during the lifetime of the Merkle tree.

[9] See Appendix A for more details about updating the `branch` array.

Complexity. Both the time and space complexity of the algorithm is linear in the tree height h. The space complexity is linear, because the size of the `branch` and `zero_hashes` arrays is h, and no other nodes are stored by the algorithm. The time complexity is also linear. For the path computation, the length of the path is h, and each node can be computed in constant time by using the two arrays. The `branch` array update can be also done in constant time as explained earlier.

Implementation and optimization. Figure 2 shows the pseudocode implementation of the incremental Merkle tree algorithm [6] that is employed in the deposit contract [14]. It consists of two main functions: `deposit` and `get_deposit_root`. The `deposit` function takes as input a new deposit hash, and inserts it into the Merkle tree. The `get_deposit_root` function computes and returns the root of the current partial Merkle tree whose leaves are filled with the deposit hashes received up to that point.

Specifically, the `deposit` function fills the first (leftmost) empty leaf node with a given deposit hash, and updates a single element of the `branch` array. The `get_deposit_root` function computes the tree root by traversing a path from the last (rightmost) non-empty leaf to the root.

As an optimization, the `deposit` function does not fully compute the path from the leaf to the root, but computes only a smaller partial path from the leaf to the node that is needed to update the `branch` array. Indeed, for all odd-numbered deposits (i.e., 1^{st} deposit, 3^{rd} deposit, $\cdots$), such a partial path is empty, because the leaf node is the one needed for the `branch` array update. In that case, the `deposit` function returns immediately in constant time. For even-numbered deposits, the partial path is not empty but still much smaller than the full path in most cases. This optimization is useful when the tree root computation is not needed for every single partial Merkle tree. Indeed, in many cases, multiple deposit hashes are inserted at once, for which only the root of the last partial Merkle tree is needed.

Correctness. Consider a Merkle tree of height h employed in the deposit contract. Suppose that a sequence of `deposit` function calls are made, say `deposit`(v_1), `deposit`(v_2), $\cdots$, and `deposit`(v_m), where $m < 2^h$. Then, the function call `get_deposit_root`() will return the root of the Merkle tree whose leaves are filled with the deposit data hashes v_1, v_2, $\cdots$, v_m, respectively, in order from left to right, starting from the leftmost one.

Note that the correctness statement requires the condition $m < 2^h$, that is, the rightmost leaf must be kept empty, which means that the maximum number of deposits that can be stored in the tree using this incremental algorithm is $2^h - 1$ instead of 2^h. See Section 5.1 for more details.

The proof of the correctness is presented in Appendix A.

Remark. Since the `deposit` function reverts when `deposit_count` $\geq 2^{\texttt{TREE\_HEIGHT}} - 1$, the loop in the `deposit` function cannot reach the last iteration, thus the loop bound (in line 17 of Figure 2) can be safely decreased to `TREE_HEIGHT` $- 1$.

```
# globals
zero_hashes: int[TREE_HEIGHT] = {0} # zero array
branch:      int[TREE_HEIGHT] = {0} # zero array
deposit_count: int = 0  # max: 2^TREE_HEIGHT - 1

fun init() -> unit:
    i: int = 0
    while i < TREE_HEIGHT - 1:
        zero_hashes[i+1] = hash(zero_hashes[i], zero_hashes[i])
        i += 1

fun deposit(value: int) -> unit:
    assert deposit_count < 2^TREE_HEIGHT - 1
    deposit_count += 1
    size: int = deposit_count
    i: int = 0
    while i < TREE_HEIGHT:
        if size % 2 == 1:
            break
        value = hash(branch[i], value)
        size /= 2
        i += 1
    branch[i] = value

fun get_deposit_root() -> int:
    root: int = 0
    size: int = deposit_count
    h: int = 0
    while h < TREE_HEIGHT:
        if size % 2 == 1: # size is odd
            root = hash(branch[h], root)
        else:             # size is even
            root = hash(root, zero_hashes[h])
        size /= 2
        h += 1
    return root
```

Fig. 2. Pseudocode implementation of the incremental Merkle tree algorithm employed in the deposit contract [14].

4 Bytecode Verification of the Deposit Contract

Now we present the formal verification of the compiled bytecode of the deposit contract. The bytecode verification ensures that the compiled bytecode is a sound refinement of the source code. This rules out the need to trust the compiler.

As illustrated in Section 2, we first manually refined the abstract model (in which we proved the algorithm correctness) to the bytecode specification (Section 4.1). For the refinement, we consulted the ABI interface standard [13] (to identify, e.g., `calldata` and `output` in the illustrating example of Section 2), as well as the bytecode (to identify, e.g., the `pc` and `stack` information).[10] Then, we used the KEVM verifier [45] to verify the compiled bytecode against the refined specification. We adopted the KEVM verifier to reason about all possible corner-case behaviors of the compiled bytecode, especially those introduced by certain unintuitive and questionable aspects of the underlying Ethereum Virtual Machine (EVM) [59]. This was possible because the KEVM verifier is derived from a complete formal semantics of the EVM, called KEVM [24]. Our formal specification and verification artifacts are publicly available at [49].

Let us elaborate on specific low-level behaviors verified against the bytecode. In addition to executing the incremental Merkle tree algorithm, most of the functions perform certain additional low-level tasks, and we verified that such tasks are correctly performed. Specifically, for example, given deposit data,[11] the `deposit` function computes its 32-byte hash (called Merkleization) according to the SimpleSerialize (SSZ) specification [18]. The leaves of the Merkle tree store only the computed hashes instead of the original deposit data. The `deposit` function also emits a `DepositEvent` log that contains the original deposit data, where the log message needs to be encoded as a byte sequence following the contract event ABI specification [13]. Other low-level operations performed by those functions that we verified include: correct zero-padding for the 32-byte alignment, correct conversions from big-endian to little-endian, input bytes of the SHA2-256 hash function being correctly constructed, and return values being correctly serialized to byte sequences according to the ABI specification [13].

We also verified a liveness property that the contract is always able to accept a new (valid) deposit as long as a sufficient amount of gas is provided. This liveness is not trivial since it needs to hold even in any future hard-fork where the gas fee schedule is changed. Indeed, we found a bug of the Vyper compiler that a hard-coded amount of gas is attached when calling to the `memcpy` builtin function (more precisely, the `ID` precompiled contract). This bug could make the deposit contract non-functional in a certain future hard-fork where the gas fee schedule for the builtin function is increased, because the contract will always fail due to the out-of-gas exception no matter how much gas users supply. This bug has been reported and fixed [44].

[10] However, we want to note that the Vyper compiler can be augmented to extract such information, which can automate the refinement process to a certain extent. We leave that as future work.

[11] Each deposit data consists of the public key, the withdrawal credentials, the deposit amount, and the signature of the deposit owner.

Our formal specification includes both positive and negative behaviors. The positive behaviors describe the desired behaviors of the contracts in a legitimate input state. The negative behaviors, on the other hand, describe how the contracts handle exceptional cases (e.g., when benign users feed invalid inputs by mistake, or malicious users feed crafted inputs to take advantage of the contracts). The negative behaviors are mostly related to security properties.

4.1 Summary of Bytecode Specification

We summarize the formal specification of the deposit contract bytecode that we verified. The full specification can be found at [48].

Constructor ***init*** updates the storage as follows:

$$\texttt{zero\_hashes}[i] \leftarrow ZH(i) \quad \text{for all } 1 \leq i < 32$$

where $ZH(i)$ is a 32-byte word that is recursively defined as follows:

$$\begin{aligned} ZH(i+1) &= \mathsf{hash}(ZH(i) \texttt{ ++ } ZH(i)) \quad \text{for } 0 \leq i < 31 \\ ZH(0) &= 0 \end{aligned}$$

where hash denotes the SHA2-256 hash function, and ++ denotes the byte concatenation.

Function ***get_deposit_count*** returns $LE_{64}(\texttt{deposit\_count})$, where $LE_{64}(x)$ denotes the 64-bit little-endian representation of x (for $0 \leq x < 2^{64}$). That is, for a given $x = \Sigma_{0 \leq i < 8}(a_i \cdot 256^i)$, $LE_{64}(x) = \Sigma_{0 \leq i < 8}(a_{7-i} \cdot 256^i)$, where $0 \leq a_i < 256$. Note that $LE_{64}(\texttt{deposit\_count})$ is always defined because of the contract invariant of $\texttt{deposit\_count} < 2^{32}$. This function does not alter the storage state.

Function ***get_deposit_root*** returns:

$$\mathsf{hash}(RT(32) \texttt{ ++ } LE_{64}(\texttt{deposit\_count}) \texttt{ ++ } 0_{[24]})$$

where $RT(32)$ is the Merkle tree root, recursively defined as follows:

$$\begin{aligned} RT(i+1) &= \left\{ \begin{array}{ll} \mathsf{hash}(\texttt{branch}[i] \texttt{ ++ } RT(i)), & \text{if } \left\lfloor \texttt{deposit\_count}/2^i \right\rfloor \text{ is odd} \\ \mathsf{hash}(RT(i) \texttt{ ++ } \texttt{zero\_hashes}[i]), & \text{otherwise} \end{array} \right\} \\ & \quad \text{for } 0 \leq i < 32 \\ RT(0) &= 0 \end{aligned}$$

and $0_{[24]}$ denotes 24 zero-bytes. This function does not alter the storage state.

Function `deposit` updates the storage state as follows:

$$\texttt{deposit\_count} \leftarrow \mathsf{old}(\texttt{deposit\_count}) + 1$$
$$\texttt{branch}[k] \leftarrow \mathit{ND}(k)$$

where $\mathsf{old}(\texttt{deposit\_count})$ denotes the value of `deposit_count` at the beginning of the function, k is the smallest integer less than 32 such that $\left\lfloor \frac{\mathsf{old}(\texttt{deposit\_count})+1}{2^k} \right\rfloor$ is odd,[12] and $\mathit{ND}(K)$ is a 32-byte word that is recursively defined as follows:

$$\mathit{ND}(i+1) = \mathsf{hash}(\texttt{branch}[i] \texttt{ ++ } \mathit{ND}(i)) \quad \text{for } 0 \leq i < 32$$

where $\mathit{ND}(0)$ denotes the deposit data root that is a Merkle proof of the deposit data that consists of the public key, the withdrawal credentials, the deposit amount, and the signature. The `deposit` function also emits a `DepositEvent` log that includes both the deposit data and the $\mathsf{old}(\texttt{deposit\_count})$ value. For the full details about the deposit data root computation and the `DepositEvent` log, refer to [48].

Negative behaviors. The contract reverts when either a call-value (i.e., `msg.value`) or a call-data (i.e., `msg.data`) is invalid. A call-value is invalid when it is non-zero but the called function is not payable (i.e., no `@payable` annotation). A call-data is invalid when its size is less than 4 bytes, or its first four bytes do not match the signature of any public functions in the contract. Note that any extra contents in the call-data are silently ignored.[13]

The `deposit` function reverts if the tree is full, the deposit amount is less than the required minimum amount, or the call-data is not well-formed. See Section 5 for more details about these negative behaviors of the `deposit` function.

5 Findings and Lessons Learned

In the course of our formal verification effort, we found subtle bugs [37,35,36] of the deposit contract, as well as a couple of refactoring suggestions [38,39,40] that can improve the code readability and reduce the gas cost. The subtle bugs of the deposit contract are partly due to bugs of the Vyper compiler [41,42,43,44] that we newly found (and reported to the Vyper team) in the verification process.

Below we elaborate on the bugs we found and lessons we learned along the way. We note that all the bugs of the deposit contract have been reported, confirmed, and properly fixed in the latest version (`v0.11.2`).

5.1 Maximum Number of Deposits

In the original version of the contract that we were asked to verify, a bug is triggered when all of the leaf nodes of a Merkle tree are filled with deposit

[12] Note that such k always exists since we have $\mathsf{old}(\texttt{deposit\_count}) < 2^{32} - 1$ by the assertion at the beginning of the function.

[13] We have not yet found an attack that can exploit this behavior.

data, in which case the contract (specifically, the `get_deposit_root` function) incorrectly computes the root hash of a tree, returning the zero root hash (i.e., the root hash of an empty Merkle tree) regardless of the content of leaf nodes. For example, suppose that we have a Merkle tree of height 2, which has four leaf nodes, and every leaf node is filled with certain deposit data, say v_1, v_2, v_3, and v_4, respectively. Then, while the correct root hash of the tree is $\mathsf{hash}(\mathsf{hash}(v_1, v_2), \mathsf{hash}(v_3, v_4))$, the `get_deposit_root` function returns $\mathsf{hash}(\mathsf{hash}(0, 0), \mathsf{hash}(0, 0))$, which is incorrect.

Due to the complex logic of the code, it is non-trivial to properly fix this bug without significantly rewriting the code, and thus we suggested a workaround that simply forces to never fill the last leaf node, i.e., accepting only $2^h - 1$ deposits at most, where h is the height of a tree. We note that, however, it is infeasible in practice to trigger this buggy behavior in the current setting, since the minimum deposit amount is 1 Ether and the total supply of Ether is less than 130M which is much smaller than 2^{32}, thus it is not feasible to fill all the leaves of a tree of height 32. Nevertheless, this bug has been fixed by the contract developers as we suggested, since the contract may be used in other settings in which the buggy behavior can be triggered and an exploit may be possible. Refer to [37] for more details.

We also want to note that this bug was quite subtle to catch. Indeed, we had initially thought that the original code was correct until we failed to write a formal proof of the correctness theorem. The failure of our initial attempt to prove the correctness led us to identify a missing premise (i.e., the correctness condition $m < 2^h$ in Section 3) that was needed for the theorem to hold, from which we could find the above buggy behavior scenario, and suggested the bugfix. This experience reconfirms the importance of formal verification. Although we were not "lucky" to find this bug when we had eyeball-reviewed the code, which is all traditional security auditors do, the formal verification process thoroughly guided and even "forced" us to find it eventually.

5.2 ABI Standard Conformance of `get_deposit_count` Function

In the previous version, the `get_deposit_count` function does not conform to the ABI standard [13], where its return value contains incorrect zero-padding [35], due to a Vyper compiler bug [41]. Specifically, in the buggy version of the compiled bytecode, the `get_deposit_count` function, whose return type is `bytes[8]`, returns a byte sequence of length 96, where the last byte is `0x20` while it should be `0x00`. According to the ABI specification [13], the last 24 bytes must be all zero, serving as zero-pad for the 32-byte alignment. Thus the return value does not conform to the ABI standard. This is problematic because any contract (written in either Solidity or Vyper) that calls to (the buggy version of) the deposit contract, expecting that the `deposit_count` function conforms to the ABI standard, could have misbehaved.[14]

[14] The returned byte sequence, including the incorrect last byte, is copied to the caller's memory. If the caller reuses the last byte assuming that it is zero, the garbage value will be passed around, which may break the business logic of the caller.

This buggy behavior is mainly due to a subtle Vyper compiler bug [41] that fails to correctly compile a function whose return type is `bytes[`n`]` where $n < 16$. This leads to the compiled function returning a byte sequence with insufficient zero-padding as mentioned above, failing to conform to the ABI standard.

We note that this bug could not have been detected if we did not take the bytecode as the verification target. This reconfirms that the bytecode-level verification is critical to ensure the ultimate correctness (unless we formally verify the underlying compiler), because we cannot (and should not) trust the compiler.

5.3 Checking Well-Formedness of Calldata

The calldata decoding process in the previous version of the compiled bytecode does not have sufficient runtime-checks for the well-formedness of calldata. As such, it fails to detect certain ill-formed calldata, causing invalid deposit data to be put into the Merkle tree. This is problematic especially when clients make mistakes and send deposit transactions with incorrectly encoded calldata, which may result in losing their deposit fund.

Specifically, we found a counter-example ill-formed calldata whose size (196 bytes) is much less than that of well-formed calldata (356 bytes). The problem, however, is that the `deposit` function does *not* reject the ill-formed calldata, but simply inserts certain invalid (garbage) deposit data in the Merkle tree. Since the invalid deposit data cannot pass the signature validation later, no one can claim the deposited fund associated with this, and the deposit owner loses the fund. Note that this happens even though the `deposit` function employs assertions at the beginning of the function that ensures the size of each of the arguments is correct, which turned out to not work as expected.

This problem would not exist if the Vyper compiler thoroughly generated runtime checks to ensure the well-formedness of calldata.[15] However, since it was not trivial to fix the compiler to generate such runtime checks, we suggested several ways to improve the deposit contract source code to prevent this behavior without fixing the compiler. After careful discussion with the deposit contract development team, we together decided to employ a checksum-based approach where the `deposit` function takes as an additional input a checksum for the deposit data, and rejects any ill-formed calldata using the checksum. The checksum-based approach is the least intrusive and the most gas-efficient of all the suggested fixes. For more details of other suggested fixes, refer to [36].

We note that this issue was found when we were verifying the negative behaviors of the deposit contract. This shows the importance of having the formal specification to include not only positive but also negative behaviors.

[15] The compiler developers failed to consider the case when the given calldata is not correctly encoded. For example, while the header of calldata contains offsets (i.e., pointers) to the positions of data elements, it could be the case that certain offsets are beyond the calldata range. In that case, the calldata can be accessed outside its bounds, due to the missing runtime-checks.

5.4 Liveness

As mentioned in Section 4, the previous version of the deposit contract fails to satisfy a liveness property in that it may not be able to accept a new deposit, even if it is valid, in a certain future hard-fork that updates the gas fee schedule. This was mainly due to another subtle Vyper compiler bug [44] that generates bytecode where a hard-coded amount of gas is supplied when calling to certain precompiled contracts. Although this hard-coded amount of gas is sufficient in the current hard-fork (code-named Istanbul [17]), it may not be sufficient in a certain future hard-fork that increases the gas fee schedule of the precompiled contracts. In such a future hard-fork, the previous version of the deposit contract will always fail due to the out-of-gas exception, regardless of how much gas is initially supplied. Refer to [44] for more details.

We admit that we could not find this issue until the deposit contract development team carefully reviewed and discussed with us the formal specification [48] of the bytecode. Initially, we considered only the behaviors of the bytecode in the current hard-fork, without identifying the requirement that the contract bytecode should work in any future hard-fork. We identified the missing requirement, and found this liveness issue, at a very late stage of the formal verification process, which delayed the completion of formal verification.

This experience essentially illustrates the well-known problem caused by the gap between the intended behaviors (that typically exists only informally) by developers, and the formal specification written by verification engineers. To reduce this gap, the two groups should work closely together, or ideally, developers should write their own specifications in the first place. For the former, the formal verification process should involve developers more frequently. For the latter, the formal verification tools should become much easier to use without requiring advanced knowledge of formal methods. We leave both as future work.

5.5 Discussion

Verification effort. The net effort for formal verification took 7 person-weeks (excluding various discussions with developers, reporting bugs and following-up, especially for compiler bugs, etc.), where the algorithm correctness proof took 2 person-weeks, and the bytecode verification took 5 person-weeks. This includes the time spent on writing specifications as well. The bytecode specification consists of ∼1,000 LOC (excluding comments), in addition to auxiliary lemmas consisting of ∼200 LOC. The size of the source code is ∼100 LOC, and the number of instructions in the compiled bytecode is ∼3,000.

The verification engineers were highly experienced, holding a doctoral degree in formal methods, with more than two years of experience in smart contract verification. The development team, however, was assumed to have no advanced knowledge of formal verification. The interactions with the development team were mostly discussions on questionable behaviors of the code, how to fix the bugs we found, and how to improve the code clarity, etc. For the specification review, we wrote a separate informal document in English so that they could review and confirm that no important properties are missed in the specification.

Trust base. The validity of the bytecode verification result assumes the correctness of the bytecode specification and the KEVM verifier. The algorithm correctness proof is partially mechanized—only the proof of major lemmas are mechanized in the K framework. The non-mechanized proofs are included in our trust base. The Vyper compiler is *not* in the trust base.

Continuous verification. The verification target contract was a moving target. Even if the contract code had been frozen before starting the formal verification process, the code (both source code or bytecode) was updated in the middle of the verification process, to fix bugs found during the process. Indeed, we found several bugs in both the contract and the compiler, and each time we found a bug, we had to re-verify the newly compiled bytecode that fixes the bug. Here the problem was the overhead of re-verification. About 20% of the bytecode verification effort was spent on re-verification.

The re-verification overhead could have been reduced by automatically adjusting formal specifications to updated bytecode, and/or making specifications as independent of the specific details of the bytecode as possible. For example, the current bytecode specification employs specific program-counter (PC) values to refer to some specific positions of the bytecode, especially when specifying loop invariants. Most of such PC values need to be updated whenever the bytecode is modified. The re-verification overhead could have been reduced by automatically updating such PC values, or even having the specification refer to specific positions without using PC values. We leave this as future work.

6 Related Work

Static analysis and verification of smart contracts. There have been proposed many static analysis tools [28,25,5,57,32,29,20,56,10] that are designed to automatically detect a certain fixed set of bugs and vulnerabilities of smart contracts, at the cost of generality and expressiveness. VerX [46] can verify past-time linear temporal properties over multiple runs of smart contracts, but it requires the target contracts to be effectively loop-free.

There also have been proposed verification tools that allow us to specify and verify arbitrary functional correctness and/or security properties, such as [3,22] based on the F* proof assistant [55], [1] based on Isabelle/HOL [33], the KEVM verifier [45] based on the K framework [51], and VeriSol [27] based on Boogie [2]. The KEVM verifier has also been used to verify high-profile and challenging smart contracts [50], including a multi-signature wallet called Gnosis Safe [21], a decentralized token exchange called Uniswap [58], and a partial consensus mechanism called Casper FFG [7].

Verification of systems software. There are many success stories of formal verification of systems software, from OS kernels [26,23,31], to file systems [8,52], to cryptographic code [4]. While most of the verified systems code is either synthesized from specifications, or implemented (or adjusted) to be verification-friendly, there also exist efforts [12,9] to verify actual production code as is. Such

efforts are necessary especially when the production code is highly performance-critical and/or existing development processes are hard to change to help produce verification-friendly code. The deposit contract we verified was given to us at the code-frozen stage, and also performance-critical (especially in terms of the gas cost), and thus we took and verified the given production-ready code as is, without any modification except for fixing bugs.

7 Conclusion

We reported our end-to-end formal verification of the Ethereum 2.0 deposit contract. We adopted the refinement-based verification approach to ensure the end-to-end correctness of the contract while minimizing the verification effort. Specifically, we first proved that the incremental Merkle tree algorithm is correctly implemented in the contract, and then verified that the compiled bytecode is correctly generated from the source code. Although we found several critical issues of the deposit contract during the formal verification process, some of which were due to subtle hidden Vyper compiler bugs, all of the issues of the deposit contract have been properly fixed in the latest version (`v0.11.2`) of the deposit contract, compiled by the Vyper compiler version `1761-HOTFIX-v0.1.0-beta.13`. We conclude that the latest deposit contract bytecode will behave as expected—as specified in the formal specification [48].

We note that this formal verification result is established without trusting the Vyper compiler, which means that the formally verified bytecode is correct even if the Vyper compiler is buggy [11]. Indeed, the Vyper compiler has been improved enough to generate a correct bytecode from the deposit contract. In other words, remaining Vyper compiler bugs, if any, have *not* been triggered when generating the specific bytecode we formally verified.

References

1. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in isabelle/hol. In: Proceedings of the 7th ACM International Conference on Certified Programs and Proofs. CPP 2018 (2018)
2. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (2005)
3. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. PLAS 2016 (2016)
4. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S.T.V., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017 (2017)
5. Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B.: Vandal: A scalable security analysis framework for smart contracts. CoRR **abs/1809.03981** (2018)
6. Buterin, V.: Progressive Merkle Tree. `https://github.com/ethereum/research/blob/master/beacon_chain_impl/progressive_merkle_tree.py`
7. Buterin, V., Griffith, V.: Casper the friendly finality gadget. CoRR **abs/1710.09437** (2017)
8. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the FSCQ file system. In: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015 (2015)
9. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous formal verification of amazon s2n. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (2018)
10. ConsenSys Diligence: MythX. `https://mythx.io/`
11. ConsenSys Diligence: Vyper Security Review. `https://diligence.consensys.net/audits/2019/10/vyper/`
12. Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (2018)
13. Ethereum Foundation: Contract ABI Specification. `https://solidity.readthedocs.io/en/v0.6.1/abi-spec.html`
14. Ethereum Foundation: Ethereum 2.0 Deposit Contract. `https://github.com/ethereum/eth2.0-specs/blob/v0.11.2/deposit_contract/contracts/validator_registration.vy`
15. Ethereum Foundation: Ethereum 2.0 Specifications. `https://github.com/ethereum/eth2.0-specs`
16. Ethereum Foundation: Ethereum Foundation Spring 2019 Update. `https://blog.ethereum.org/2019/05/21/ethereum-foundation-spring-2019-update/`

17. Ethereum Foundation: Hardfork Meta: Istanbul. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1679.md
18. Ethereum Foundation: SimpleSerialize (SSZ). https://github.com/ethereum/eth2.0-specs/tree/dev/ssz
19. Ethereum Foundation: Vyper. https://vyper.readthedocs.io
20. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019 (2019)
21. Gnosis Ltd.: Gnosis Safe. https://safe.gnosis.io/
22. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. In: Proceedings of the 7th International Conference on Principles of Security and Trust. POST 2018 (2018)
23. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: Certikos: An extensible architecture for building certified concurrent OS kernels. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016 (2016)
24. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Moore, B., Zhang, Y., Park, D., Ştefănescu, A., Roşu, G.: Kevm: A complete semantics of the ethereum virtual machine. In: Proceedings of the 31st IEEE Computer Security Foundations Symposium. CSF 2018 (2018)
25. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium. NDSS 2018 (2018)
26. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009 (2009)
27. Lahiri, S.K., Chen, S., Wang, Y., Dillig, I.: Formal specification and verification of smart contracts for azure blockchain. CoRR **abs/1812.08829** (2018)
28. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS 2016 (2016)
29. Marescotti, M., Blicha, M., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Computing exact worst-case gas consumption for smart contracts. In: Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV (2018)
30. Merkle, R.C.: A digital signature based on a conventional encryption function. In: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology. CRYPTO '87 (1988)
31. Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., Wang, X.: Hyperkernel: Push-button verification of an OS kernel. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017 (2017)
32. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018 (2018)

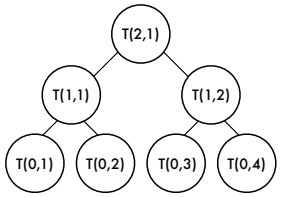
33. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)
34. NIST: Perfect Binary Tree. https://xlinux.nist.gov/dads/HTML/perfectBinaryTree.html
35. Park, D.: Ethereum 2.0 Deposit Contract Issue 1341: Non ABI-standard return value of get_deposit_count of deposit contract. https://github.com/ethereum/eth2.0-specs/issues/1341
36. Park, D.: Ethereum 2.0 Deposit Contract Issue 1357: Ill-formed calldata to deposit contract can add invalid deposit data. https://github.com/ethereum/eth2.0-specs/issues/1357
37. Park, D.: Ethereum 2.0 Deposit Contract Issue 26: Maximum deposit count. https://github.com/ethereum/deposit_contract/issues/26
38. Park, D.: Ethereum 2.0 Deposit Contract Issue 27: Redundant assignment in init(). https://github.com/ethereum/deposit_contract/issues/27
39. Park, D.: Ethereum 2.0 Deposit Contract Issue 28: Loop fusion optimization. https://github.com/ethereum/deposit_contract/issues/28
40. Park, D.: Ethereum 2.0 Deposit Contract Issue 38: A refactoring suggestion for the loop of deposit(). https://github.com/ethereum/deposit_contract/issues/38
41. Park, D.: Vyper Issue 1563: Insufficient zero-padding bug for functions returning byte arrays of size < 16. https://github.com/vyperlang/vyper/issues/1563
42. Park, D.: Vyper Issue 1599: Off-by-one error in zero_pad(). https://github.com/vyperlang/vyper/issues/1599
43. Park, D.: Vyper Issue 1610: Non-semantics-preserving refactoring for zero_pad(). https://github.com/vyperlang/vyper/issues/1610
44. Park, D.: Vyper Issue 1761: Potentially insufficient gas stipend for precompiled contract calls. https://github.com/vyperlang/vyper/issues/1761
45. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A Formal Verification Tool for Ethereum VM Bytecode. In: Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2018 (2018)
46. Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: VerX: Safety Verification of Smart Contracts. https://files.sri.inf.ethz.ch/website/papers/sp20-verx.pdf
47. Podelski, A., Rybalchenko, A.: Transition invariants. In: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science. LICS 2004 (2004)
48. Runtime Verification, Inc.: Bytecode Behavior Specification of Ethereum 2.0 Deposit Contract. https://github.com/runtimeverification/verified-smart-contracts/blob/master/deposit/bytecode-verification/deposit-spec.ini.md
49. Runtime Verification, Inc.: Formal Verification of Ethereum 2.0 Deposit Contract. https://github.com/runtimeverification/verified-smart-contracts/tree/master/deposit
50. Runtime Verification, Inc.: Formally Verified Smart Contracts. https://github.com/runtimeverification/verified-smart-contracts
51. Serbanuta, T., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Rosu, G.: The K primer (version 3.3). Electr. Notes Theor. Comput. Sci. **304**, 57–80 (2014)
52. Sigurbjarnarson, H., Bornholt, J., Torlak, E., Wang, X.: Push-button verification of file systems via crash refinement. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016 (2016)

53. Stefanescu, A., Ciobaca, S., Mereuta, R., Moore, B.M., Serbanuta, T., Rosu, G.: All-Path Reachability Logic. Logical Methods in Computer Science **15**(2) (2019)
54. Stefanescu, A., Park, D., Yuwen, S., Li, Y., Rosu, G.: Semantics-Based Program Verifiers for All Languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2016 (2016)
55. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Béguelin, S.Z.: Dependent types and multi-monadic effects in F. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016 (2016)
56. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: Static analysis of ethereum smart contracts. In: 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018 (2018)
57. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018 (2018)
58. Uniswap: Uniswap Exchange Protocol. `https://uniswap.io/`
59. Wood, G.: Ethereum: A Secure Decentralised Generalised Transaction Ledger. `https://ethereum.github.io/yellowpaper/paper.pdf`

A Formalization and Correctness Proof of the Incremental Merkle Tree Algorithm

We formalize the incremental Merkle tree algorithm [6], especially the one employed in the deposit contract [14], and prove its correctness w.r.t. the original full-construction Merkle tree algorithm [30].

Notations. Let T be a perfect binary tree [34] (i.e., every node has exactly two child nodes) of height h, and $T(l, i)$ denote its node at level l and index i, where the level of leaves is 0, and the index of the left-most node is 1. For example, if $h = 2$, then $T(2, 1)$ denotes the root whose children are $T(1, 1)$ and $T(1, 2)$, and the leaves are denoted by $T(0, 1)$, $T(0, 2)$, $T(0, 3)$, and $T(0, 4)$, as follows:



We write $[\![T(l, i)]\!]$ to denote the value of the node $T(l, i)$, but we omit $[\![\cdot]\!]$ when the meaning is clear in the context.

Let us define two functions, $\uparrow$ and $\upharpoonleft$, as follows:

$$\uparrow x = \lceil x/2 \rceil \tag{1}$$

$$\upharpoonleft x = \lfloor x/2 \rfloor \tag{2}$$

Moreover, let us define $\uparrow^k x = \uparrow (\uparrow^{k-1} x)$ for $k \geq 2$, $\uparrow^1 x = \uparrow x$, and $\uparrow^0 x = x$. Let $\{T(k, \uparrow^k x)\}_{k=0}^{h}$ be a path $\{T(0, \uparrow^0 x), T(1, \uparrow^1 x), T(2, \uparrow^2 x), \cdots, T(h, \uparrow^h x)\}$. We write $\{T(k, \uparrow^k x)\}_k$ if h is clear in the context. Let us define $\upharpoonleft^k$ and $\{T(k, \upharpoonleft^k x)\}_k$ similarly. For the presentation purpose, let $T(l, 0)$ denote a dummy node which has the parent $T(l+1, 0)$ and the children $T(l-1, 0)$ and $T(l-1, 1)$. Note that, however, these dummy nodes are only conceptual, allowing the aforementioned paths to be well-defined, but *not* part of the tree at all.

In this notation, for a non-leaf, non-root node of index i, its left child index is $2i - 1$, its right child index is $2i$, and its parent index is $\uparrow i$. Also, note that $\{T(k, \uparrow^k m)\}_k$ is the path starting from the m-th leaf going all the way up to the root.

First, we show that two paths $\{T(k, \uparrow^k x)\}_k$ and $\{T(k, \upharpoonleft^k (x-1))\}_k$ are parallel with a "distance" of 1.

Lemma 1. *For all $x \geq 1$, and $k \geq 0$, we have:*

$$(\uparrow^k x) - 1 = \upharpoonleft^k (x - 1) \tag{3}$$

Proof. Let us prove by induction on k. When $k = 0$, we have $(\uparrow^0 x) - 1 = x - 1 = \upharpoonleft^0 (x - 1)$. When $k = 1$, we have two cases:

– When x is odd, that is, $x = 2y + 1$ for some $y \geq 0$:

$$(\uparrow x) - 1 = (\uparrow (2y+1)) - 1 = \left\lceil \frac{2y+1}{2} \right\rceil - 1 = y = \left\lfloor \frac{2y}{2} \right\rfloor = {}^{\circlearrowleft} 2y = {}^{\circlearrowleft} (x-1)$$

– When x is even, that is, $x = 2y$ for some $y \geq 1$:

$$(\uparrow x) - 1 = (\uparrow 2y) - 1 = \left\lceil \frac{2y}{2} \right\rceil - 1 = y - 1 = \left\lfloor \frac{2y-1}{2} \right\rfloor = {}^{\circlearrowleft} (2y-1) = {}^{\circlearrowleft} (x-1)$$

Thus, we have:

$$(\uparrow x) - 1 = {}^{\circlearrowleft} (x-1) \tag{4}$$

Now, assume that (3) holds for some $k = l \geq 1$. Then,

$$\begin{aligned}
\uparrow^{l+1} x &= \uparrow (\uparrow^{l} x) && \text{(By the definition of } \uparrow^{k}) \\
&= \uparrow (({}^{\circlearrowleft^{l}} (x-1)) + 1) && \text{(By the assumption)} \\
&= ({}^{\circlearrowleft} ({}^{\circlearrowleft^{l}} (x-1))) + 1 && \text{(By Equation 4)} \\
&= {}^{\circlearrowleft^{l+1}} (x-1) + 1 && \text{(By the definition of } {}^{\circlearrowleft^{k}})
\end{aligned}$$

which concludes.

Now let us define the Merkle tree.

Definition 1. *A perfect binary tree T of height h is a* Merkle tree *[30], if the leaf node contains data, and the non-leaf node's value is the hash of its children's, i.e.,*

$$\forall 0 < l \leq h.\ \forall 0 < i \leq 2^{h-l}.\ T(l,i) = \mathsf{hash}(T(l-1, 2i-1), T(l-1, 2i)) \tag{5}$$

Let T_m be a partial Merkle tree up-to m *whose first m leaves contain data and the other leaves are zero, i.e.,*

$$T_m(0,i) = 0 \quad \textit{for all } m < i \leq 2^h \tag{6}$$

Let Z be the zero Merkle tree *whose leaves are all zero, i.e., $Z(0,i) = 0$ for all $0 < i \leq 2^h$. That is, $Z = T_0$. Since all nodes at the same level have the same value in Z, we write $Z(l)$ to denote the value at the level l, i.e., $Z(l) = Z(l,i)$ for any $0 < i \leq 2^{h-l}$.*

Now we formulate the relationship between the partial Merkle trees. Given two partial Merkle trees T_{m-1} and T_m, if their leaves agree up-to $m-1$, then they only differ on the path $\{T_m(k, \uparrow^k m)\}_k$. This is formalized in Lemma 2.

Lemma 2. *Let T_m be a partial Merkle tree up-to $m > 0$ of height h, and let T_{m-1} be another partial Merkle tree up-to $m-1$ of the same height. Suppose their leaves agree up to $m-1$, that is, $T_{m-1}(0,i) = T_m(0,i)$ for all $1 \leq i \leq m-1$. Then, for all $0 \leq l \leq h$, and $1 \leq i \leq 2^{h-l}$,*

$$T_{m-1}(l,i) = T_m(l,i) \quad \textit{when } i \neq \uparrow^{l} m \tag{7}$$

Proof. Let us prove by induction on l. When $l = 0$, we immediately have $T_{m-1}(0,i) = T_m(0,i)$ for any $i \neq m$ by the premise and Equation 6. Now, assume that (7) holds for some $l = k$. Then by Equation 5, we have $T_{m-1}(k+1,i) = T_m(k+1,i)$ for any $i \neq \uparrow (\uparrow^k m) = \uparrow^{k+1} m$, which concludes.

Corollary 1 induces a *linear-time* incremental Merkle tree insertion algorithm [6].

Corollary 1. *T_m can be constructed from T_{m-1} by computing only $\{T_m(k, \uparrow^k m)\}_k$, the path from the new leaf, $T_m(0,m)$, to the root.*

Proof. By Lemma 2.

Let us formulate more properties of partial Merkle trees.

Lemma 3. *Let T_m be a partial Merkle tree up-to m of height h, and Z be the zero Merkle tree of the same height. Then, for all $0 \leq l \leq h$, and $1 \leq i \leq 2^{h-l}$,*

$$T_m(l,i) = Z(l) \quad \textit{when } i >\uparrow^l m \tag{8}$$

Proof. Let us prove by induction on l. When $l = 0$, we immediately have $T_m(0,i) = Z(0) = 0$ for any $m < i \leq 2^h$ by Equation 6. Now, assume that (8) holds for some $0 \leq l = k < h$. First, for any $i \geq (\uparrow^{k+1} m) + 1$, we have:

$$2i - 1 \geq (2 \uparrow^{k+1} m) + 1 = 2 \left\lceil \frac{\uparrow^k m}{2} \right\rceil + 1 \geq 2\frac{\uparrow^k m}{2} + 1 = (\uparrow^k m) + 1 \tag{9}$$

Then, for any $\uparrow^{k+1} m < i \leq 2^{h-(k+1)}$, we have:

$$\begin{aligned} T_m(k+1,i) &= \mathsf{hash}(T_m(k,2i-1), T_m(k,2i)) && \text{(By Equation 5)} \\ &= \mathsf{hash}(Z(k), Z(k)) && \text{(By Equations 8 and 9)} \\ &= Z(k+1) && \text{(By the definition of } Z\text{)} \end{aligned}$$

which concludes.

Lemma 4 induces a *linear-space* incremental Merkle tree insertion algorithm.

Lemma 4. *A path $\{T_m(k, \uparrow^k m)\}_k$ can be computed by using only two other paths, $\{T_{m-1}(k, \curvearrowleft^k (m-1)))\}_k$ and $\{Z(k)\}_k$.*

Proof. We will construct the path from the leaf, $T_m(0,m)$, which is given. Suppose we have constructed the path up to $T_m(q, \uparrow^q m)$ for some $q > 0$ by using only two other sub-paths, $\{T_{m-1}(k, \curvearrowleft^k (m-1))\}_{k=0}^{q-1}$ and $\{Z(k)\}_{k=0}^{q-1}$. Then, to construct $T_m(q+1, \uparrow^{q+1} m)$, we need the sibling of $T_m(q, \uparrow^q m)$, where we have two cases:

- Case $(\uparrow^q m)$ is odd. Then, we need the right-sibling $T_m(q, (\uparrow^q m)+1)$, which is $Z(q)$ by Lemma 3.

– Case $(\uparrow^q m)$ is even. Then, we need the left-sibling $T_m(q, (\uparrow^q m) - 1)$, which is $T_m(q, \hookleftarrow^q (m-1))$ by Lemma 1, which is in turn $T_{m-1}(q, \hookleftarrow^q (m-1))$ by Lemma 2.

By the mathematical induction on k, we conclude.

Lemma 5. *Let $h =$ `TREE_HEIGHT`. For any integer $0 \leq m < 2^h$, the two paths $\{T_m(k, \hookleftarrow^k m)\}_k$ and $\{T_{m+1}(k, \hookleftarrow^k (m+1))\}_k$ always converge, that is, there exists unique $0 \leq l \leq h$ such that:*

$$(\hookleftarrow^k m) + 1 = \hookleftarrow^k (m+1) \text{ is even for all } 0 \leq k < l \tag{10}$$

$$(\hookleftarrow^k m) + 1 = \hookleftarrow^k (m+1) \text{ is odd for } k = l \tag{11}$$

$$\hookleftarrow^k m = \hookleftarrow^k (m+1) \text{ for all } l < k \leq h \tag{12}$$

$$T_m(k, \hookleftarrow^k m) = T_{m+1}(k, \hookleftarrow^k (m+1)) \text{ for all } l < k \leq h \tag{13}$$

Proof. Equation 12 follows from Equation 11, since for an odd integer x, $\hookleftarrow (x - 1) = \hookleftarrow x$. Also, Equation 13 follows from Lemma 2, since $\uparrow^k (m+1) = (\hookleftarrow^k m) + 1 \neq \hookleftarrow^k m = \hookleftarrow^k (m+1)$ by Lemma 1 and Equation 12. Thus, we only need to prove the unique existence of l satisfying (10) and (11). The existence of l is obvious since $1 \leq m + 1 \leq 2^h$, and one can find the smallest l satisfying (10) and (11). Now, suppose there exist two different $l_1 < l_2$ satisfying (10) and (11). Then, $\hookleftarrow^{l_1} (m+1)$ is odd since l_1 satisfies (11), while $\hookleftarrow^{l_1} (m+1)$ is even since l_2 satisfies (10), which is a contradiction, thus l is unique, and we conclude.

A.1 Pseudocode

Figure 2 shows the pseudocode of the incremental Merkle tree algorithm [6] that is employed in the deposit contract [14]. It maintains a global counter `deposit_count` to keep track of the number of deposits made, and two global arrays `zero_hashes` and `branch`, which corresponds to Z (Definition 1) and a certain part of $\{T_m(k, \hookleftarrow^k m)\}_k$, where m denotes the value of `deposit_count`. The `init` function is called once at the beginning to initialize `zero_hashes` which is never updated later. The `deposit` function inserts a given new leaf value in the tree by incrementing `deposit_count` and updating only a single element of `branch`. The `get_deposit_root` function computes the root of the current partial Merkle tree T_m.

Since the loops are bounded to the tree height and the size of global arrays is equal to the tree height, it is clear that both time and space complexities of the algorithm are linear.

A.2 Correctness Proof

Now we prove the correctness of the incremental Merkle tree algorithm shown in Figure 2.

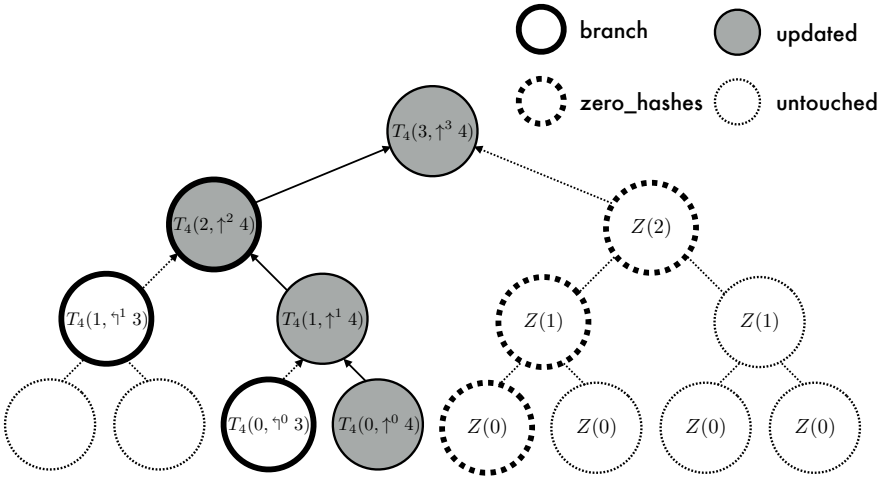


Fig. 3. A partial Merkle tree T_4 of height 3, illustrating the incremental Merkle tree algorithm shown in Figure 2, where `TREE_HEIGHT` $= 3$. The bold-lined nodes correspond to the `branch` array. The bold-dotted-lined nodes correspond to the `zero_hashes` array. The `get_deposit_root` function computes the gray nodes by using only the bold-lined nodes (i.e., `branch`) and the bold-dotted-lined nodes (i.e., `zero_hashes`), where `deposit_count` $= 4$.

Theorem 1 (Correctness of Incremental Merkle Tree Algorithm). *Suppose that the* ***init*** *function is executed at the beginning, followed by a sequence of* ***deposit*** *function calls, say* ***deposit***(v_1), ***deposit***(v_2), $\cdots$, *and* ***deposit***(v_m), *where* $m < 2^{\texttt{TREE\_HEIGHT}}$. *Then, the function call* ***get_deposit_root***$()$ *will return the root of the partial Merkle tree* T_m *such that* $T_m(0, i) = v_i$ *for all* $1 \leq i \leq m$.

Proof. By Lemmas 6, 7, 8, and 9.

Note that the correctness theorem requires the condition $m < 2^h$, where h is the tree height, that is, the rightmost leaf must be kept empty, which means that the maximum number of deposits that can be stored in the tree using this incremental algorithm is $2^h - 1$ instead of 2^h. See Section 5.1 for more details.

Lemma 6 (`init`). *Once* ***init*** *is executed,* ***zero_hashes*** *denotes* Z, *that is,*

$$\textbf{\textit{zero\_hashes}}[k] = Z(k) \tag{14}$$

for $0 \leq k <$ ***TREE_HEIGHT***.

Proof. By the implementation of `init` and the definition of Z in Definition 1.

Lemma 7 (`deposit`). *Suppose that, before executing* ***deposit***, *we have:*

$$\textbf{\textit{deposit\_count}} = m < 2^{\texttt{TREE\_HEIGHT}} - 1 \tag{15}$$

$$\textbf{\textit{branch}}[k] = T_m(k, \leftarrow^k m) \quad \textit{if } \leftarrow^k m \textit{ is odd} \tag{16}$$

Then, after executing $\texttt{deposit}(v)$*, we have:*

$$\texttt{deposit\_count}' = m + 1 \leq 2^{\texttt{TREE\_HEIGHT}} - 1 \tag{17}$$

$$\texttt{branch}'[k] = T_{m+1}(k, \natural^k\ (m+1)) \quad \textit{if } \natural^k\ (m+1) \textit{ is odd} \tag{18}$$

for any $0 \leq k < \texttt{TREE\_HEIGHT}$*, where:*

$$T_{m+1}(0, m+1) = v \tag{19}$$

Proof. Let $h = \texttt{TREE\_HEIGHT}$. Equation 17 is obvious by the implementation of `deposit`. Let us prove Equation 18. Let l be the unique integer described in Lemma 5. We claim that `deposit` updates only $\texttt{branch}[l]$ to be $T_{m+1}(l, \natural^l\ (m+1))$. Then, for all $0 \leq k < l$, $\natural^k\ (m+1)$ is not odd. For $k = l$, we conclude by the aforementioned claim. For $l < k \leq h$, we conclude by Equation 13 and the fact that $\texttt{branch}[k]$ is not modified (by the aforementioned claim).

Now, let us prove the aforementioned claim. Since `branch` is updated only at line 23, we only need to prove $\texttt{i} = l$ and $\texttt{value} = T_{m+1}(l, \natural^l\ (m+1))$ at that point. We claim the following loop invariant at line 17:

$$\texttt{i} = i < \texttt{TREE\_HEIGHT} \tag{20}$$

$$\texttt{value} = T_{m+1}(i, \natural^i\ (m+1)) \tag{21}$$

$$\texttt{size} = \natural^i\ (m+1) \tag{22}$$

$$\natural^k\ (m+1) \text{ is even for any } 0 \leq k < i \tag{23}$$

Note that i cannot reach $\texttt{TREE\_HEIGHT}$, since $(m+1) < 2^{\texttt{TREE\_HEIGHT}}$. Thus, by the loop invariant, we have the following after the loop at line 23:

$$\texttt{i} = i < \texttt{TREE\_HEIGHT} \tag{24}$$

$$\texttt{value} = T_{m+1}(i, \natural^i\ (m+1)) \tag{25}$$

$$\texttt{size} = \natural^i\ (m+1) \text{ is odd} \tag{26}$$

$$\natural^k\ (m+1) \text{ is even for any } 0 \leq k < i \tag{27}$$

Moreover, by Lemma 5, we have $i = l$, which suffices to conclude the aforementioned claim.

Now we only need to prove the loop invariant. First, at the beginning of the first iteration, we have $i = 0$, $\texttt{value} = v = T_{m+1}(0, m+1)$ by (19), and $\texttt{size} = (m+1)$, which satisfies the loop invariant. Now, assume that the invariant holds at the beginning of the i^{th} iteration that does not reach the `break` statement at line 19 (i.e., $\texttt{size} = \natural^i\ (m+1)$ is even). Then, $\texttt{i}' = i + 1$, $\texttt{size}' = \natural^{i+1}\ (m+1)$, and:

$$\begin{aligned}
T_{m+1}(i+1, \natural^{i+1}\ (m+1)) &= \mathsf{hash}(T_{m+1}(i, \natural^i\ m), T_{m+1}(i, \natural^i\ (m+1))) \\
&\qquad \text{(by Equation 10)} \\
&= \mathsf{hash}(T_m(i, \natural^i\ m), \texttt{value}) \\
&\qquad \text{(by Lemmas 1 \& 2 and Equation 21)} \\
&= \mathsf{hash}(\texttt{branch}[i], \texttt{value}) \quad \text{(by Equations 16 \& 10)} \\
&= \texttt{value}'
\end{aligned}$$

Thus, the loop invariant holds at the beginning of the $(i+1)^{\text{th}}$ iteration as well, and we conclude.

Lemma 8 (Contract Invariant). *Let $m =$ `deposit_count`. Then, once `init` is executed, the following contract invariant holds. For all $0 \leq k <$ `TREE_HEIGHT`,*

1. *`zero_hashes`$[k] = Z(k)$*
2. *`branch`$[k] = T_m(k, \lnot^k m)$ if $\lnot^k m$ is odd*
3. *`deposit_count`$\leq 2^{\texttt{TREE\_HEIGHT}} - 1$*

Proof. Let us prove each invariant item.

1. By Lemma 6, and the fact that `zero_hashes` is updated by only `init`.
2. By Lemma 7, and the fact that `branch` is updated by only `deposit`.
3. By the assertion of `deposit` (at line 13 of Figure 2), and the fact that `deposit_count` is updated by only `deposit`.

Lemma 9 (`get_deposit_root`). *The `get_deposit_root` function computes the path $\{T_m(k, \uparrow^k (m+1))\}_k$ and returns the root $T_m(h, 1)$, given a Merkle tree T_m of height h, that is, `deposit_count` $= m < 2^h$ and `TREE_HEIGHT` $= h$ when `get_deposit_root` is invoked.*

Proof. We claim the following loop invariant at line 29, which suffices to conclude the main claim.

$$\begin{aligned} \texttt{h} &= k \quad \text{where } 0 \leq k \leq h \\ \texttt{size} &= \lnot^k m \\ \texttt{root} &= T_m(k, \uparrow^k (m+1)) \end{aligned}$$

Now let us prove the above loop invariant claim by the mathematical induction on k. The base case ($k = 0$) is trivial, since $\lnot^0 m = m$, $\uparrow^0 (m+1) = m+1$, and $T_m(0, m+1) = 0$ by Definition 1. Assume that the loop invariant holds for some $k = l$. Let $\texttt{h}'$, $\texttt{size}'$, and $\texttt{root}'$ denote the values at the next iteration $k = l+1$. Obviously, we have $\texttt{h}' = l+1$ and $\texttt{size}' = \lnot^{l+1} m$. Also, we have $(\lnot^l m) + 1 = \uparrow^l (m+1)$ by Lemma 1. Now, we have two cases:

- Case $\texttt{size} = \lnot^l m$ is odd. Then, $\uparrow^l (m+1)$ is even. Thus,

$$\begin{aligned} T_m(l+1, \uparrow^{l+1} (m+1)) &= \mathsf{hash}(T_m(l, \lnot^l m), T_m(l, \uparrow^l (m+1))) \\ &= \mathsf{hash}(\texttt{branch}[l], \texttt{root}) \qquad \text{(by Lemma 8)} \\ &= \texttt{root}' \end{aligned}$$

- Case $\texttt{size} = \lnot^l m$ is even. Then, $\uparrow^l (m+1)$ is odd. Thus,

$$\begin{aligned} T_m(l+1, \uparrow^{l+1} (m+1)) &= \mathsf{hash}(T_m(l, \uparrow^l (m+1)), T_m(l, (\uparrow^l (m+1)) + 1)) \\ &= \mathsf{hash}(\texttt{root}, Z(l)) \qquad \text{(by Lemma 3)} \\ &= \mathsf{hash}(\texttt{root}, \texttt{zero\_hashes}[l]) \qquad \text{(by Lemma 8)} \\ &= \texttt{root}' \end{aligned}$$

Thus, we have $\texttt{root}' = T_m(l+1, \uparrow^{l+1} (m+1))$, which concludes.

Mechanized Proofs. The loop invariant proofs of Lemma 7 and Lemma 9 are mechanized in the K framework, which can be found at [49].