End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract
(Solidity Reimplementation)

Daejun Park, Yi Zhang, and Grigore Rosu

Runtime Verification, Inc.

Abstract. We present our formal verification of the deposit smart contract, reimplemented in Solidity, whose correctness is critical for the security of Ethereum 2.0, a new Proof-of-Stake protocol for the Ethereum blockchain. The deposit contract employs an incremental Merkle tree algorithm whose correctness is highly nontrivial, and had not been proved before. We have formally verified the correctness of the compiled bytecode of the deposit contract to avoid the need to trust the underlying compiler. No critical issues were found in the Solidity implementation of the deposit contract.

1 Introduction

The deposit smart contract [6] is a gateway to join Ethereum 2.0 [7] 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.[1] 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 Solidity [10],[2] employs the Merkle tree [13] 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 cost significantly, the contract implements an *incremental Merkle tree algorithm* [3]. The

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

[2] Initially, the contract had been written in Vyper [11] but later it was reimplemented in Solidity due to the concerns [4] regarding the Vyper compiler. Indeed, we had also formally verified the previous Vyper implementation, which can be found at [21].

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.[3] 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.

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.

Verification Target. The specific target of our formal verification is the latest version (`2ac62c4`) of the deposit contract written in Solidity, provided that the contract is compiled by the Solidity compiler `v0.6.8` with the optimization enabled (`-optimize-runs 5000000`).

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 [22,24,16], 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; }
```

[3] This was done as part of a contract funded by the Ethereum Foundation [8].

Given this program, we first manually write an abstract model of the program in the K framework [22]. Such a K model is essentially a state transition system of the program, and can be written as follows:

$$\begin{array}{l}
\texttt{rule: sum(}n\texttt{)} \Rightarrow \texttt{loop(s: }0\texttt{, i: }1\texttt{, n: }n\texttt{)} \\
\texttt{rule: loop(s: }s\texttt{, i: }i\texttt{, n: }n\texttt{)} \Rightarrow \texttt{loop(s: }s+i\texttt{, i: }i+1\texttt{, n: }n\texttt{) when } i \leq n \\
\texttt{rule: loop(s: }s\texttt{, i: }i\texttt{, n: }n\texttt{)} \Rightarrow \texttt{return(}s\texttt{) when } i > n
\end{array}$$

These transition rules correspond to the initialization, the `while` loop, and the return statement, respectively. The indexed tuple $\texttt{(s: }s\texttt{, i: }i\texttt{, n: }n\texttt{)}$ represents the state of the program variables `s`, `i`, and `n`.[4]

Then, given the abstract model, we specify the functional correctness property in reachability logic [23], as follows:

$$\texttt{claim: sum(}n\texttt{)} \Rightarrow \texttt{return(}\tfrac{n(n+1)}{2}\texttt{) when } n > 0$$

This reachability claim says that $\texttt{sum(}n\texttt{)}$ 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 [24], which requires us only to provide the following loop invariant:[5]

$$\texttt{invariant: loop(s: }\tfrac{i(i-1)}{2}\texttt{, i: }i\texttt{, n: }n\texttt{)} \Rightarrow \texttt{return(}\tfrac{n(n+1)}{2}\texttt{) when } 0 < i \leq n+1$$

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:

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

Here, the indexed tuple `evm(pc:_, calldata:_, stack:_, output:_)` represents (part of) the Ethereum Virtual Machine (EVM) state, and $\texttt{\#bytes(}N\texttt{,}V\texttt{)}$ 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 [24,16]. 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:

[4] 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).

[5] 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 [24, Section 4]. These loop invariants can be also seen as transition invariants [17].

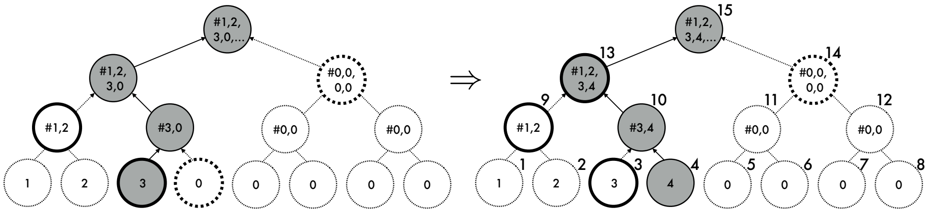


Fig. 1. Illustration of the incremental Merkle tree algorithm. The left tree is updated to the right tree by inserting a new data hash in the fourth leaf (node 4). Only the path from the new leaf to the root (i.e., the gray nodes) are computed by the algorithm (hence linear-time). The bold-lined (and 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.

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

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

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 [24,16] and the formal semantics of EVM [12] 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 [13] is a perfect binary tree [14] 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 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.[6] 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.

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 [3] that is employed in the deposit contract [6]. 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

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

```
# 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 constructor() -> 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 [6].

leaf to the node that is needed to update the `branch` array. Indeed, for all odd-numbered deposits (i.e., 1st deposit, 3rd 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.

The proof of the correctness is presented in Appendix A.

Remark 1. 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` $- 1$.

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 [5] (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).[7] Then, we used the KEVM verifier [16] 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) [26]. This was possible because the KEVM verifier is derived

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

from a complete formal semantics of the EVM, called KEVM [12]. Our formal specification and verification artifacts are publicly available at [20].

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,[8] the `deposit` function computes its 32-byte hash (called Merkleization) according to the SimpleSerialize (SSZ) specification [9]. 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 [5]. 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 [5].

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 [19].

Constructor function `constructor` updates the storage as follows:

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

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

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

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

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

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

Function ***`get_deposit_root`*** returns:

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

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

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

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

Function ***`deposit`*** updates the storage state as follows:

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

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,[9] and $\mathit{ND}(K)$ is a 32-byte word that is recursively defined as follows:

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

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 [19].

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.[10]

The `deposit` function reverts if the tree is full, the deposit amount is less than the required minimum amount or not a multiple of Gwei (10^9 wei), or the call-data is not well-formed.

[9] 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.

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

Table 1. Gas analysis of the deposit contract. The total gas cost is the summation of the tx cost, the opcode cost, and the memory cost. The multiple rows denote the lower and upper bounds of the gas cost.

	Total Gas Cost	Tx Cost	Opcode Cost	Memory Cost
`constructor`	770,144	53,000	716,470	674
`supportsInterface`	21,461	21,192	254	15
	21,526	21,240	271	15
`get_deposit_count`	23,278	21,064	2,187	27
`get_deposit_root`	98,259	21,064	76,849	346
	98,579	21,064	77,169	346
`deposit`	46,162	22,812	23,252	98
	157,642	25,308	131,929	405

4.2 Gas Analysis

Table 1 shows the theoretical gas analysis of the deposit contract, essentially summarizing the gas cost description of the bytecode specification [19]. It provides the lower and upper bounds of the gas cost for each external function, meaning that the amount of gas consumed by any possible execution falls within the bounds.[11] Each gas cost is broken down into three categories: the tx cost (the base fee of sending a transaction, plus the calldata fee), the opcode cost (the gas fee of executing instructions), and the memory cost (the gas fee of using memory). The calldata fee is calculated based on the number of zero and non-zero bytes in the given calldata, where the gas fee per zero byte is smaller than that of non-zero byte. Thus the calldata fee varies for the same function unless the function takes no argument.

- The gas cost of the `constructor` function is constant because it is executed only once when the contract is deployed, taking no argument.
- The gas cost of the `get_deposit_count` functions is also constant, because it has only a single execution path and it takes no argument.[12]
- The `supportsInterface` function has two execution paths due to the short-circuit evaluation of the logical-or (`||`) operation, causing two different opcode costs. The calldata fee varies depending on the number of zero and non-zero bytes in the given argument—the minimum is when the argument is `0x00000000` and the maximum is when it is `0xffffffff`.
- The `get_deposit_root` function has 2^{32} execution paths, because of the `if` branches inside the loop of 32 iterations. The minimum opcode cost is when only the `else` branch is taken in all iterations of the loop (i.e., when `deposit_count` $= 0$), and the maximum is when only the `then` branch is taken (i.e., when `deposit_count` $= 2^{32} - 1$).

[11] For the simplicity, here we consider only the normal execution that does not revert in the middle. The gas cost for reverting executions is smaller than that of normal executions.

[12] We assume that the calldata is well-formed, and does not include any garbage bytes.

- The `deposit` function has 32 (non-reverting) execution paths, because of the conditional `return` statement inside the loop of 32 iterations. Both the opcode and memory costs are proportional to the number of loop iterations.[13] Another factor for the opcode cost is the variable gas fee of the storage update [25]. The calldata cost varies depending on the arguments.

5 Discussion

The deposit contract had been initially written in Vyper, and the Vyper-compiled bytecode had been formally verified [21]. The Vyper language was chosen because of the security-first language design that prioritizes security and auditability over efficiency and convenience for developers. However, the community concerns [4,2] about the current Vyper compiler motivated the deposit contract developers to reimplement the contract in Solidity with re-verifying the new Solidity-compiled bytecode. The underlying motivation is primarily based on the maturity of the Solidity compiler and its wider adoption in the Ethereum community than the Vyper compiler.

To minimize the overhead of reimplementation followed by re-verification of the new deposit contract, the Solidity implementation is designed to keep as much as possible the overall structure of the Vyper implementation. However, the Solidity implementation still made several substantial improvements over the original Vyper implementation that do not change the original interface of the contract and incur no significant overhead in the re-verification. The improvements made in the Solidity implementation include the following:

- The `to_little_endian_64` function has been rewritten to improve both readability and performance. While in the Vyper implementation, a loop was employed to reorder the (least significant) eight bytes of the argument, in the Solidity implementation, the loop has been unrolled using an advanced feature of Solidity to succinctly and intuitively represent the reordering operation. (See Figure 3 for the comparison of the two implementations.) Moreover, with the optimization enabled, the Solidity compiler has been able to generate quite gas-efficient bytecode from the source code.
- The EIP-165 [18] support has been added. The EIP-165 is a protocol for identifying the interfaces implemented in a given contract. Supporting the EIP-165 protocol, the new deposit contract allows other Dapps to query what interfaces it implements.
- Message strings for the `require` statements have been added to help users to easily identify the reasons for failures that may occur while using the deposit contract.
- An additional `require` statement has been added to ensure that the deposit amount is a multiple of Gwei (10^9 wei). The unit of deposit is Gwei and

[13] The memory cost can be made constant by improving the memory footprint [15].

```
function
to_little_endian_64(... val) ... {
  ret = new bytes(8);
  bytes8 bytesValue = bytes8(val);
  ret[0] = bytesValue[7];
  ret[1] = bytesValue[6];
  ret[2] = bytesValue[5];
  ret[3] = bytesValue[4];
  ret[4] = bytesValue[3];
  ret[5] = bytesValue[2];
  ret[6] = bytesValue[1];
  ret[7] = bytesValue[0];
}
```

```
def
to_little_endian_64(val: ...) ...:
  y: uint256 = 0
  x: uint256 = val
  for _ in range(8):
    y = shift(y, 8)
    y = y + bitwise_and(x, 255)
    x = shift(x, -8)
  return slice(convert(y ...) ...)
```

Fig. 3. Comparison of the `to_little_endian_64` function in the Solidity (left) and Vyper (right) implementations.

the incoming value is divided by it, the remainder being lost. The additional `require` statement avoids lost remainders due to mistakes made by users.[14]

- As mentioned in Remark 1, the contract contains an infeasible (dead) path. As a defensive programming measure, an `assert(false)` statement has been added at the end of the infeasible path.

Despite the additional runtime checks, the Solidity implementation is more gas-efficient (30∼40% less gas cost) than the original Vyper implementation, thanks to the advanced code optimization of the Solidity compiler. The Solidity-generated bytecode, however, still has room to further improve the gas efficiency, e.g., duplicate address sanitation [1] and non-optimal memory footprint in the loop [15]. Nonetheless, the gas inefficiency issues are not critical, and do not compromise the safety of the contract.

Overall, the Solidity implementation and the compiled bytecode are in a better shape than the original Vyper implementation, thanks to the aforementioned improvements, and the advantage of the Solidity compiler.

6 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. No critical bugs were found in the

[14] In the current gas price, the gas cost for the additional `require` statement is more than the potentially saved amount of ether ($\leq$ 1 Gwei). However, the gas price may change in the future, and more importantly, the additional check allows users to detect early any potential mistake, which has its own value.

Solidity implementation of the deposit contract.[15] We conclude that the latest deposit contract (`2ac62c4`) will behave as expected—as specified in the formal specification [19], provided that the contract is compiled by the Solidity compiler `v0.6.8` with the optimization enabled (`-optimize-runs 5000000`).

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 Solidity compiler is *not* in the trust base.

[15] Several issues found in the previous Vyper implementation have already been fixed in the Solidity reimplementation. See [21] for more details of the previous findings.

References

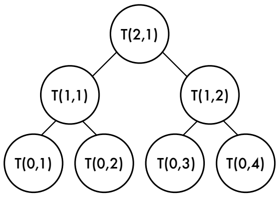
1. Beregszaszi, A.: Solidity Issue 8670. https://github.com/ethereum/solidity/issues/8670
2. Bugrara, S.: A Review of the Deposit Contract. https://docs.google.com/document/d/1sbsMvngo0kBfzy8fbhJ55kl6F1tUxmECMUuK75F5H9E
3. Buterin, V.: Progressive Merkle Tree. https://github.com/ethereum/research/blob/master/beacon_chain_impl/progressive_merkle_tree.py
4. ConsenSys Diligence: Vyper Security Review. https://diligence.consensys.net/audits/2019/10/vyper/
5. Ethereum Foundation: Contract ABI Specification. https://solidity.readthedocs.io/en/v0.6.1/abi-spec.html
6. Ethereum Foundation: Ethereum 2.0 Deposit Contract. https://github.com/axic/eth2-deposit-contract/blob/r1/deposit_contract.sol
7. Ethereum Foundation: Ethereum 2.0 Specifications. https://github.com/ethereum/eth2.0-specs
8. Ethereum Foundation: Ethereum Foundation Spring 2019 Update. https://blog.ethereum.org/2019/05/21/ethereum-foundation-spring-2019-update/
9. Ethereum Foundation: SimpleSerialize (SSZ). https://github.com/ethereum/eth2.0-specs/tree/master/ssz
10. Ethereum Foundation: Solidity. https://solidity.readthedocs.io
11. Ethereum Foundation: Vyper. https://vyper.readthedocs.io
12. 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)
13. 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)
14. NIST: Perfect Binary Tree. https://xlinux.nist.gov/dads/HTML/perfectBinaryTree.html
15. Park, D.: Solidity Issue 9046. https://github.com/ethereum/solidity/issues/9046
16. 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)
17. Podelski, A., Rybalchenko, A.: Transition invariants. In: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science. LICS 2004 (2004)
18. Reitwießner, C., Johnson, N., Vogelsteller, F., Baylina, J., Feldmeier, K., Entriken, W.: ERC-165 Standard Interface Detection. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-165.md
19. Runtime Verification, Inc.: Formal Verification of Deposit Contract Bytecode. https://github.com/runtimeverification/deposit-contract-verification/tree/master/bytecode-verification
20. Runtime Verification, Inc.: Formal Verification of Ethereum 2.0 Deposit Contract. https://github.com/runtimeverification/deposit-contract-verification
21. Runtime Verification, Inc.: Formal Verification of Ethereum 2.0 Deposit Contract (Vyper Implementation). https://github.com/runtimeverification/verified-smart-contracts/tree/master/deposit

22. 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)
23. 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)
24. 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)
25. Tang, W.: EIP 2200: Structured Definitions for Net Gas Metering. `https://eips.ethereum.org/EIPS/eip-2200`
26. 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 [3], especially the one employed in the deposit contract [6], and prove its correctness w.r.t. the original full-construction Merkle tree algorithm [13].

Notations. Let T be a perfect binary tree [14] (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 $\Lsh$, as follows:

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

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

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 $\Lsh^k$ and $\{T(k, \Lsh^k x)\}_k$ 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 $\{T(k, \Lsh^k (x-1))\}_k$ are parallel with a "distance" of 1.

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

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

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

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

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

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

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

Thus, we have:

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

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

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

which concludes.

Now let us define the Merkle tree.

Definition 1. *A perfect binary tree T of height h is a* Merkle tree *[13], 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 [3].

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, $\{T_{m-1}(k, \Lsh^k (m-1)))\}_k$ 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, $\{T_{m-1}(k, \Lsh^k (m-1))\}_{k=0}^{q-1}$ 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 $T_m(q, \Lsh^q (m-1))$ by Lemma 1, which is in turn $T_{m-1}(q, \Lsh^q (m-1))$ 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 $\{T_m(k, \Lsh^k m)\}_k$ and $\{T_{m+1}(k, \Lsh^k (m+1))\}_k$ always converge, that is, there exists unique $0 \leq l \leq h$ such that:*

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

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

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

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

Proof. Equation 12 follows from Equation 11, since for an odd integer x, $\Lsh (x-1) = \Lsh x$. Also, Equation 13 follows from Lemma 2, since $\uparrow^k (m+1) = (\Lsh^k m) + 1 \neq \Lsh^k m = \Lsh^k (m+1)$ 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, $\Lsh^{l_1} (m+1)$ is odd since l_1 satisfies (11), while $\Lsh^{l_1} (m+1)$ 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 [3] that is employed in the deposit contract [6]. 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 $\{T_m(k, \Lsh^k m)\}_k$, where m denotes the value of `deposit_count`. The `constructor` 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.

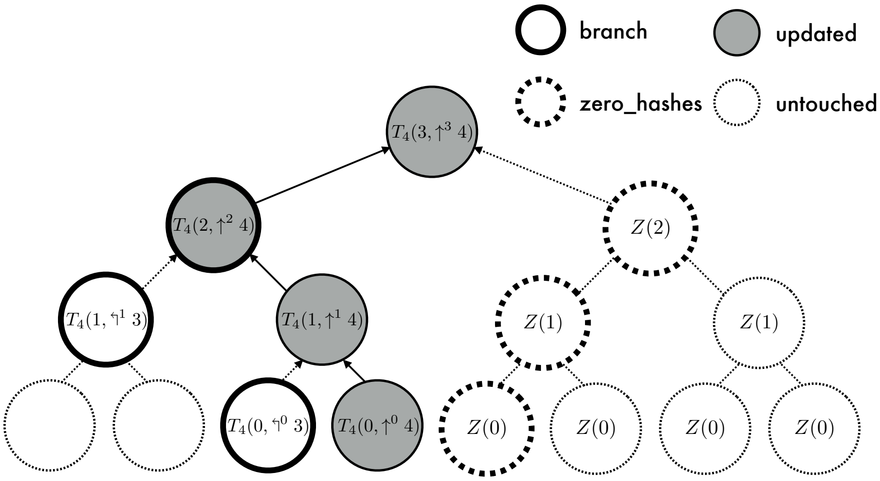


Fig. 4. A partial Merkle tree T_4 of height 3, illustrating the incremental Merkle tree algorithm shown in Figure 2, where `TREE_HEIGHT` $= 3$. The bold-lined nodes correspond to the `branch` array. The bold-dotted-lined nodes correspond to the `zero_hashes` array. The `get_deposit_root` function computes 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* `constructor` *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.

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

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

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:*

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

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

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

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

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

for any $0 \leq k <$ ***TREE_HEIGHT****, where:*

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

Proof. Let h = `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 $\mathtt{branch}[l]$ to be $T_{m+1}(l, \Lsh^{l} (m+1))$. Then, for all $0 \leq k < l$, $\Lsh^{k} (m+1)$ 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 $\mathtt{branch}[k]$ 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 $\mathtt{i} = l$ and $\mathtt{value} = T_{m+1}(l, \Lsh^{l} (m+1))$ at that point. We claim the following loop invariant at line 17:

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

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

$$\mathtt{size} = \Lsh^{i} (m+1) \tag{22}$$

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

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

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

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

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

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

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 $i = 0$, $\mathtt{value} = v = T_{m+1}(0, m+1)$ by (19), and $\mathtt{size} = (m+1)$, 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., $\mathtt{size} = \Lsh^{i} (m+1)$ is even). Then, $\mathtt{i}' = i + 1$, $\mathtt{size}' = \Lsh^{i+1} (m+1)$, and:

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

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* $m = \texttt{deposit\_count}$*. Then, once* `init` *is executed, the following contract invariant holds. For all* $0 \leq k < \texttt{TREE\_HEIGHT}$*,*

1. $\texttt{zero\_hashes}[k] = Z(k)$
2. $\texttt{branch}[k] = T_m(k, \Lsh^k m)$ *if* $\Lsh^k m$ *is odd*
3. $\texttt{deposit\_count} \leq 2^{\texttt{TREE\_HEIGHT}} - 1$

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* $\{T_m(k, \upharpoonleft^k (m+1))\}_k$ *and returns the root* $T_m(h, 1)$*, given a Merkle tree* T_m *of height* h*, that is,* $\texttt{deposit\_count} = m < 2^h$ *and* $\texttt{TREE\_HEIGHT} = h$ *when* `get_deposit_root` *is invoked.*

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

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

Now let us prove the above loop invariant claim by the mathematical induction on k. The base case ($k = 0$) is trivial, since $\Lsh^0 m = m$, $\upharpoonleft^0 (m+1) = m+1$, 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 $\texttt{size}' = \Lsh^{l+1} m$. Also, we have $(\Lsh^l m) + 1 = \upharpoonleft^l (m+1)$ by Lemma 1. Now, we have two cases:

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

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

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

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

Thus, we have $\texttt{root}' = T_m(l+1, \upharpoonleft^{l+1} (m+1))$, 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 [20].