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

Daejun Park and Yi Zhang

Runtime Verification, Inc.

1 Introduction

Ethereum 2.0 [14] is a new sharded Proof-of-Stake (PoS) protocol that, at its early stage, lives in parallel with the existing Proof-of-Work (PoW) chain, called Eth1 chain. While the Eth1 chain is powered by miners, the new PoS chain, called Beacon chain, will be driven by validators. In the Beacon chain, the role of validators is to create and validate new blocks. The consensus protocol guarantees (under certain mild assumptions) the desired safety and liveness properties as long as a two-thirds majority of validators faithfully follow the protocol when creating and validating new blocks. The set of validators is dynamic, meaning that new validators can join and existing validators can exit over time. To be a new validator, one needs to deposit a certain amount of Ether, as a "stake", by sending a transaction (over the Eth1 network) to a designated smart contract, called deposit contract. The deposit contract records the history of deposits, which is retrieved by the Beacon chain to maintain the dynamic validator set.[1]

The deposit contract [13], written in Vyper [17], employs the Merkle tree [18] 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. Here the Merkle tree employed in the contract is very large—a Merkle tree of height 32, which can store up to 2^{32} deposits, is implemented in the contract. 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 complexities, thus saving the gas cost significantly, the contract implements an incremental Merkle tree algorithm [22]. The 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. Specifically, the algorithm maintains two arrays of length h, and each reconstruction of an updated tree requires to compute only a path starting from the new leaf (i.e., the new deposit) to the root, where the computation of the path requires only the two arrays, thus achieving the linear time and space complexities in the height of a tree.

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

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 validators who drive the entire PoS chain of Ethereum 2.0. Considering the utmost importance of the deposit contract, formal verification is demanded to ultimately guarantee its correctness.

In this report, we present our formal verification of the deposit contract. 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.[2]

We adopt the well-known 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 algorithm 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.

2 Formalization and Correctness Proof of Incremental Merkle Tree Algorithm

We formalize the incremental Merkle tree algorithm [22], especially the one employed in the deposit contract [13], and prove its correctness w.r.t. the original full-construction Merkle tree algorithm [18].

Notations Let T be a perfect binary tree [19] (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 leafs 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 leafs are denoted by $T(0, 1)$, $T(0, 2)$, $T(0, 3)$, and $T(0, 4)$, as shown in Figure 1. 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.

[2] Indeed, we found several critical bugs [8,9,10,11] of the Vyper compiler in the process of formal verification. See Section 4 for more details.

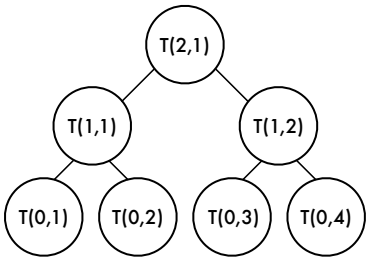

Fig. 1. A Merkle tree of height 2. We write $T(l, i)$ to denote the node of a tree T at the level l and the index i, where the level of leafs is 0, and the index of the left-most node is 1.

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 $y \geq 0$:

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

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

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

Thus, we have:

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

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

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

which concludes.

Now let us define the Merkle tree.

Definition 1. *A perfect binary tree T of height h is a* Merkle tree *[18], 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 leafs contain data and the other leafs 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 leafs 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 leafs 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 the equation (6). Now, assume that (7) holds for some $l = k$. Then by the 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 [22].

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 the 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:

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

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, $\{T_{m-1}(k, \nwarrow^k (m-1)))\}_k$ 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, $\{T_{m-1}(k, \nwarrow^k (m-1))\}_{k=0}^{q-1}$ 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 $T_m(q, \nwarrow^q (m-1))$ by Lemma 1, which is in turn $T_{m-1}(q, \nwarrow^q (m-1))$ by Lemma 2.

By the mathematical induction on k, we conclude.

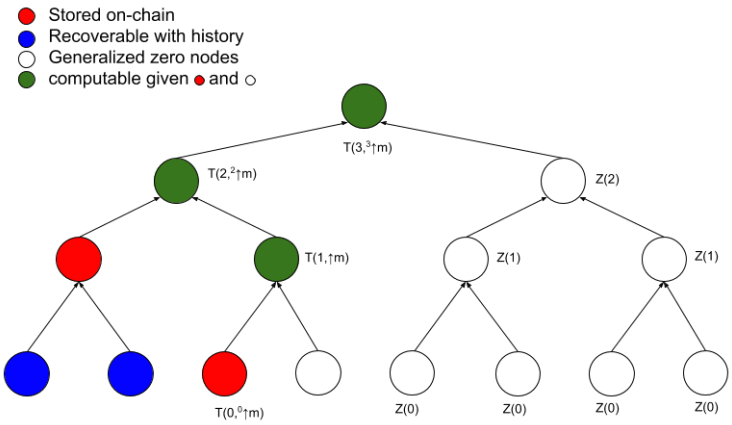

Fig. 2. A partial Merkle tree T_m of height 3, where $m = 3$, illustrating the incremental Merkle tree algorithm shown in Figure 3, where `TREE_HEIGHT` $= 3$. The red nodes correspond to the `branch` array. The white nodes are collectively represented by the `zero_hashes` array. The `get_deposit_root` function computes the green nodes by using only the red nodes (i.e., `branch`) and the white nodes (i.e., `zero_hashes`), where `deposit_count` $= 3$.

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

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

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

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

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

Proof. The equation 12 follows from the equation 11, since for an odd integer x, ${}^{\backslash}\, (x-1) = {}^{\backslash}\, x$. Also, the equation 13 follows from Lemma 2, since ${}^{\uparrow k}\, (m+1) = ({}^{\backslash k}\, m) + 1 \neq {}^{\backslash k}\, m = {}^{\backslash k}\, (m+1)$ by Lemma 1 and the 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, ${}^{\backslash l_1}\, (m+1)$ is odd since l_1 satisfies (11), while ${}^{\backslash l_1}\, (m+1)$ is even since l_2 satisfies (10), which is contradiction, thus l is unique, and we conclude.

2.1 Pseudocode

Figure 3 shows the pseudocode of the incremental Merkle tree algorithm [22] that is employed in the deposit contract [13]. 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 $\{T_m(k, \urcorner^k\ m)\}_k$, 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.

2.2 Correctness Proof

Now we prove the correctness of the incremental Merkle tree algorithm shown in Figure 3.

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 4.1 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:*

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

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

Then, after executing ***deposit***(v), *we have:*

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

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

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

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

```
# 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. 3. Pseudocode implementation of the incremental Merkle tree algorithm employed in the deposit contract [13].

Proof. Let $h = \texttt{TREE\_HEIGHT}$. The equation 17 is obvious by the implementation of `deposit`. Let us prove the equation 18. Let l be the unique integer described in Lemma 5. We claim that `deposit` updates only $\texttt{branch}[l]$ to be $T_{m+1}(l, \curvearrowleft^l (m+1))$. Then, for all $0 \leq k < l$, $\curvearrowleft^k (m+1)$ is not odd. For $k = l$, we conclude by the aforementioned claim. For $l < k \leq h$, we conclude by the 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 $\texttt{value} = T_{m+1}(l, \curvearrowleft^l (m+1))$ at that point. We claim the following loop invariant at line 17:

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

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

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

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

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

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

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

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

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

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., $\texttt{size} = \curvearrowleft^i (m+1)$ is even). Then, $\texttt{i}' = i+1$, $\texttt{size}' = \curvearrowleft^{i+1} (m+1)$, and:

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

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. $\texttt{zero\_hashes}[k] = Z(k)$
2. $\texttt{branch}[k] = T_m(k, \upharpoonleft^k m)$ *if* $\upharpoonleft^k m$ *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 3), 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,* $\texttt{deposit\_count} = m < 2^h$ *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.

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

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

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

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

- Case $\texttt{size} = \upharpoonleft^l m$ 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 [21].

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 3) can be safely decreased to `TREE_HEIGHT` $- 1$.

3 Bytecode Verification of Deposit Contract

Now we verify that the compiled bytecode is a correct refinement of the source code of the deposit contract. Especially, we verified that the pseudocode implementation of the incremental Merkle tree algorithm (as shown in Figure 3) is faithfully refined in the bytecode using the bisimulation proof method. This refinement proof rules out the need to trust the Vyper compiler.

3.1 Verified Bytecode Behaviors

We verified the functional correctness of the bytecode. For each public function, we verified that its return value and its storage state update, if any, are correct. Also, we carefully specified and verified its byte manipulation behavior, e.g., return values being correctly serialized to byte sequences according to the contract ABI specification [12], correct zero-padding for the 32-byte alignment, correct conversion from big-endian to little-endian, event log data being correctly encoded according to the contract ABI specification, input bytes of the SHA2-256 hash function being correctly constructed, and deposit data hashes (called Merkleization) being correctly computed according to the SimpleSerialize (SSZ) specification [16].

We also verified the 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 hardfolk 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 hardfolk 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 [11].

Below we summarize the verified behaviors of the deposit contract bytecode. It 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.

For the full specification of the verified bytecode behaviors, refer to [20].

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:

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

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 $0 \le x < 2^{64}$). That is, for a given $x = \Sigma_{0 \le i < 8}(a_i \cdot 256^i)$, $LE_{64}(x) = \Sigma_{0 \le i < 8}(a_{7-i} \cdot 256^i)$, where $0 \le a_i < 256$. 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 Merklee tree root, recursively defined as follows:

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

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

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

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

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,[3] and $ND(K)$ is a 32-byte word that is recursively defined as follows:

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

where $ND(0)$ 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 [20].

[3] 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.

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.[4]

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 4 for more details about these negative behaviors of the `deposit` function.

4 Findings

In the course of our formal verification effort, we found subtle bugs [4,2,3] of the deposit contract, which has been fixed in the latest version, as well as a couple of refactoring suggestions [5,6,7] that can improve the code readability and reduce the gas cost. The subtle bugs of the deposit contract are partly due to another hidden bugs of the Vyper compiler [8,9,10,11] that we revealed in the verification process.

Below we elaborate on the bugs we found. We note that all the bugs of the deposit contract have been reported, confirmed, and properly fixed in the latest version (`v0.10.0`).

4.1 Maximum Number of Deposits

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

[4] We have not yet found any attack that can exploit this behavior.

which the buggy behavior can be triggered and an exploit may be possible. Refer to [4] 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 condition $m < 2^{\texttt{TREE\_HEIGHT}}$ in Theorem 1) 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.

4.2 ABI Standard Conformance of `get_deposit_count` Function

In the previous version, the `get_deposit_count` function does not conform to the ABI standard [12], where its return value contains incorrect zero-padding, due to a Vyper compiler bug [8]. Specifically, in the buggy version of the compiled bytecode, the `get_deposit_count` function, whose return type is `bytes[8]`, returns the following 96 bytes (in hexadecimal notation):

```
0x0000000000000000000000000000000000000000000000000000000000000020
  0000000000000000000000000000000000000000000000000000000000000008
  deadbeefdeadbeef0000000000000000000000000000000000000000000000 20
```

where the first 32 bytes (in the first line) denote the offset (32 = `0x20`) to the byte array, the second 32 bytes (in the second line) denote the size of the byte array (8 = `0x8`), and the next 8 bytes "`deadbeefdeadbeef`" (at the beginning of the third line) denotes the actual content of the return byte array (of type `bytes[8]`), i.e., the (little-endian) byte representation of the deposit count. Here the problem is that the last byte (underlined) is `0x20` while it should be `0x00`. According to the ABI specification [12], the last 24 bytes must be all zero, serving as zero-pad for the 32-byte alignment. Thus the above return value does not conform to the ABI standard, which 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. This has been reported and fixed in the latest version [2].

This buggy behavior is mainly due to a subtle Vyper compiler bug [8] that fails to correctly compile a function whose return type is `bytes[`n`]` where $n < 16$, which leads to the compiled function returning an incorrect byte sequence that does not conform to the ABI standard, having insufficient zero-padding as shown above. For more details of the Vyper compiler bug, refer to [8].

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.

4.3 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.

For example, consider the following ill-formed calldata for the `deposit` function (in hexadecimal notation):

```
0xc47e300d
  0000000000000000000000000000000000000000000000000000000000000060
  0000000000000000000000000000000000000000000000000000000000000080
  00000000000000000000000000000000000000000000000000000000000000a0
  0000000000000000000000000000000000000000000000000000000000000030
  0000000000000000000000000000000000000000000000000000000000000020
  0000000000000000000000000000000000000000000000000000000000000060
```

The first four bytes (in the first line) denote the signature hash of the `deposit` function, and the remaining bytes correspond to the tuple of three arguments, `pubkey`, `withdrawal_credentials`, and `signature`. This calldata, however, is clearly ill-formed thus not valid, simply because the size (196 bytes) is much less than that of valid calldata (356 bytes).[5] The problem, however, is that the `deposit` function does *not* reject the above 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 contains the following length-checking assertions at the beginning of the function, which is quite unintuitive:

```
assert len(pubkey) == PUBKEY_LENGTH
assert len(withdrawal_credentials) == WITHDRAWAL_CREDENTIALS_LENGTH
assert len(signature) == SIGNATURE_LENGTH
```

This problem would not exist if the Vyper compiler thoroughly generated runtime checks to ensure the well-formedness of calldata. However, since it was not trivial to fix the compiler to generate such runtime checks, we suggested several ways to improve the deposit contract 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 most non-intrusive and the most gas-efficient of all the suggested fixes. For more details of other suggested fixes, refer to [3].

[5] Refer to [12] for more details of calldata well-formedness.

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 behaviors but also negative ones.

4.4 Liveness

The previous version of the deposit contract fails to satisfy the liveness property in that it may not be able to accept a new deposit, even if it is valid, in a certain future hardfork that updates the gas fee schedule. This was mainly due to another subtle Vyper compiler bug [11] 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 hardfork (codenamed Istanbul [15]), it may not be sufficient in a certain future hardfork that increases the gas fee schedule of the precompiled contracts. In such a future hardfork, the previous version of the deposit contract will always fail due to the out-of-gas exception, regardless of how much amount of gas is initially supplied. This issue has been fixed in the latest version. Refer to [11] for more details.

We admit that we could not find this issue until the deposit contract development team carefully reviewed and discussed with us about the formal specification [20] of the bytecode. Initially, we considered only the behaviors of the bytecode in the current hardfork, without identifying the requirement that the contract bytecode should work in any future hardfork. We identified the missing requirement, and found this liveness issue, at a very late stage of the formal verification process. Because of that, we did not have enough time to follow the standard process of fixing compiler-introduced bugs, that is, waiting for the next compiler release that fixes the bugs, re-generating the bytecode using the new compiler version, and re-verifying the newly generated bytecode. In order to speed up the process, we asked the Vyper compiler team to release a custom hotfix version (named `1761-HOTFIX-v0.1.0-beta.13`) to minimize the time to wait for the bugfix release. Employing the custom hotfix version also minimized the time to re-verify the newly generated bytecode, because the difference between the previous bytecode and the newly generated bytecode becomes minimal, involving only changes for fixing the bug.

This anecdotal 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, they should be in the same team. 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.

5 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 (`v0.10.0`) 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 (more precisely, as specified in the formal specification [20]).

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 [1]. 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. ConsenSys Diligence: Vyper Security Review. `https://diligence.consensys.net/audits/2019/10/vyper/`
2. Daejun Park: 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`
3. Daejun Park: 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`
4. Daejun Park: Ethereum 2.0 Deposit Contract Issue 26: Maximum deposit count. `https://github.com/ethereum/deposit_contract/issues/26`
5. Daejun Park: Ethereum 2.0 Deposit Contract Issue 27: Redundant assignment in init(). `https://github.com/ethereum/deposit_contract/issues/27`
6. Daejun Park: Ethereum 2.0 Deposit Contract Issue 28: Loop fusion optimization. `https://github.com/ethereum/deposit_contract/issues/28`
7. Daejun Park: Ethereum 2.0 Deposit Contract Issue 38: A refactoring suggestion for the loop of deposit(). `https://github.com/ethereum/deposit_contract/issues/38`
8. Daejun Park: Vyper Issue 1563: Insufficient zero-padding bug for functions returning byte arrays of size < 16. `https://github.com/vyperlang/vyper/issues/1563`
9. Daejun Park: Vyper Issue 1599: Off-by-one error in zero_pad(). `https://github.com/vyperlang/vyper/issues/1599`
10. Daejun Park: Vyper Issue 1610: Non-semantics-preserving refactoring for zero_pad(). `https://github.com/vyperlang/vyper/issues/1610`
11. Daejun Park: Vyper Issue 1761: Potentially insufficient gas stipend for precompiled contract calls. `https://github.com/vyperlang/vyper/issues/1761`
12. Ethereum Foundation: Contract ABI Specification. `https://solidity.readthedocs.io/en/v0.6.1/abi-spec.html`
13. Ethereum Foundation: Ethereum 2.0 Deposit Contract. `https://github.com/ethereum/eth2.0-specs/blob/v0.10.0/deposit_contract/contracts/validator_registration.vy`

14. Ethereum Foundation: Ethereum 2.0 Specifications. https://github.com/ethereum/eth2.0-specs
15. Ethereum Foundation: Hardfork Meta: Istanbul. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1679.md
16. Ethereum Foundation: SimpleSerialize (SSZ). https://github.com/ethereum/eth2.0-specs/tree/dev/ssz
17. Ethereum Foundation: Vyper. https://vyper.readthedocs.io
18. 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. pp. 369–378. CRYPTO '87, Springer-Verlag, London, UK, UK (1988), http://dl.acm.org/citation.cfm?id=646752.704751
19. NIST: Perfect Binary Tree. https://xlinux.nist.gov/dads/HTML/perfectBinaryTree.html
20. 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
21. Runtime Verification, Inc.: Formal Verification of Ethereum 2.0 Deposit Contract. https://github.com/runtimeverification/verified-smart-contracts/tree/master/deposit
22. Vitalik Buterin: Progressive Merkle Tree. https://github.com/ethereum/research/blob/master/beacon_chain_impl/progressive_merkle_tree.py