Verification of the Incremental Merkle Tree Algorithm with Dafny

Franck Cassez [0000−0002−4317−5025]

ConsenSys
franck.cassez@consensys.net

Abstract. The Deposit Smart Contract (DSC) is an instrumental component of the Ethereum 2.0 Phase 0 infrastructure. We have developed the first machine-checkable version of the incremental Merkle tree algorithm used in the DSC. We present our new and original correctness proof of the algorithm along with the Dafny machine-checkable version. The main results are: 1) a new proof of total correctness; 2) a software artefact with the proof in the form of the complete Dafny code base and 3) new provably correct optimisations of the algorithm.

1 Introduction

Blockchain-based decentralised platforms process transactions between parties and record them in an immutable distributed ledger. Those platforms were once limited to handle simple transactions but the next generation of platforms will routinely run *decentralised applications* (DApps) that enable users to make complex transactions (sell a car, a house or more broadly, swap assets) without the need for an institutional or governmental trusted third-party.

Smart Contracts. More precisely, the transactions are *programmatically* performed by *programs* called *smart contracts*. If there are real advantages having smart contracts act as third-parties to process transactions, there are also lots of risks that are inherent to computer programs: they can contain *bugs*. Bugs can trigger runtime errors like *division by zero* or *array-out-of-bounds*. In a networked environment these types of vulnerabilities can be exploited by malicious attackers over the network to disrupt or take control of the computer system. Other types of bugs can also compromise the business logic of a system, e.g., an implementation may contain subtle errors (e.g., using a `+=` operator in C instead of `=+`) that make them deviate from the initial intended specifications.

Unfortunately it is extremely hard to guarantee that programs and henceforth smart contracts implement the correct business logics, that they are free of common runtime errors, or that they never run into a non-terminating computation.[1] There are notorious examples of smart contract vulnerabilities that have

[1] In the Ethereum ecosystem, programs can only use a limited amount of resources, determined by the *gas limit*. So one could argue that non-terminating computations

been exploited and publicly reported: in 2016, a *reentrance* vulnerability in the Decentralised Autonomous Organisation (DAO) smart contract was exploited to steal more than USD50 Million. There may be several non officially reported similar attacks that have resulted in the loss of assets.

The Deposit Smart Contract in Ethereum 2.0. The next generation of Ethereum-based networks, Ethereum 2.0, features a new *proof-of-stake* consensus protocol. Instead of miners used in Ethereum 1.x, the new protocol relies on *validators* to create and *validate* blocks of transactions that are added to the ledger. The protocol is designed to be fault-tolerant to up to 2/3 of Byzantine (i.e., malicious or dishonest) validators. To discourage validators to deviate from an honest behaviour, they have to *stake* some assets in Ether (a crypto-currency), and if they are dishonest they can be *slashed* and lose (part of) their stake. The process of staking is handled by the *Deposit Smart Contract (DSC)*: a validator sends a transaction ("stake some Ether") by *calling* the DSC. The DSC has a *state* and can update/record the history of deposits that have occurred so far.

As a result the DSC is a mission-critical component of Ethereum 2.0, and any errors/crashes could result in inaccurate tracking of the deposits or downtime which in turn may compromise the integrity/availability of the whole system.

This could be mitigated if the DSC was a simple piece of code, but, for performance reasons, it relies on sophisticated data structures and algorithms to maintain the list of deposits so that they can be communicated over the network efficiently: the history of deposits is summarised as a unique number, a *hash*, computed using a *Merkle* (or *Hash*) tree. The tree is built incrementally using the so-called *incremental Merkle tree algorithm*, and as stated in [21]:

> *"The efficient incremental algorithm leads to the DSC implementation being unintuitive, and makes it non-trivial to ensure its correctness."*

Related Work. In this context, it is not surprising that substantial efforts, auditing, reviewing [3], testing and formal verification [20,21] has been invested to guarantee the reliability and integrity (e.g., resilience to potential attacks) of the DSC. The DSC has been the focus of an end-to-end analysis [21], including the bytecode[2] that is executed on the Ethereum Virtual Machine (EVM). However, the *incremental Merkle tree algorithm* has not been *mechanically verified* yet, even though a pen and paper proof has been proposed [20] and *partially* mechanised using the K-framework [5]. An example of the limitations of the mechanised part of the proof in [20] is that it does not contain a formal (K-)definition of Merkle trees. The mechanised sections (lemmas 7 and 9) pertain to some invariants of the algorithm but not to a proper correctness specification based on Merkle trees. The K-framework and KEVM, the formalisation of the EVM in

---

are not problematic as they cannot arise: when the gas limit is reached a computation is aborted and has no side effects. It follows that a non-terminating computation (say an infinite loop due to a programming error) combined with a finite gas limit will abort and will result in the system being unable to successfully process some or all *valid* transactions and this is a serious issue.

[2] A limitation is that the bytecode is proved using a non-trusted manual specification.

K, has been used to analyse a number of other smart contracts [25]. There are several techniques and tools[3] e.g., [8,9,1,6,28], for auditing and analysing smart contracts written in Solidity (a popular language to write Ethereum smart contracts) or EVM bytecode, but they offer limited capabilities to verify complex functional requirements.

Interesting properties of incremental Merkle trees were established in [19] using the MONA prover. This work does not prove the algorithms in the DSC which are designed to minimise gas consumption and hence split into parts: insert a value in a tree, and compute the root hash. Moreover, some key lemmas in the proofs could not be discharged by MONA.

The gold standard in program correctness is a complete logical proof that can be *mechanically checked* by a prover. This is the problem we address in this paper: to design a *machine-checkable proof* for the DSC algorithms (not the bytecode) using the Dafny language and verifier. The DSC has been deployed in November 2020. To the best of our knowledge, our analysis, completed in October 2020, provided the first fully mechanised proof that the code logic was correct, and free of runtime errors. There seem to be few comparable case-studies of Dafny-verified (or other verification-aware programming languages like Whiley [23]) code bases. The most notorious and complex one is probably the IronFleet/IronClad [10] distributed system, along with some non-trivial algorithms like DPLL [2] or Red-Black trees [24], or operating systems, FreeRTOS scheduler [16], and ExpressOS [15]. Other proof assistants like Coq [22], Isabelle/HOL [18] or Lean [17] have also been extensively used to write machine-checkable proofs of algorithms [12,27,7,26] and software systems [11,14].

Our Contribution. We present a thorough analysis of the incremental Merkle tree algorithm used in the DSC. Our results are available as software artefacts, written using the CAV-awarded Dafny[4] verification-aware programming language [13]. This provides a self-contained machine checkable and reproducible proof of the DSC algorithms. Our contribution is many-fold and includes:

- a *new original simple proof* of the incremental Merkle tree algorithm. In contrast to the previous non-mechanised proof in [20] we do not attempt to directly prove the existing algorithm, but rather to *design* and refine it. Our proof is *parametric* in the height of the tree, and *hash* functions;
- a *logical specification* using a formal definition of Merkle trees, and a *new functional version* of the algorithm that is proved correct against this specification; the functional version is used to specify the invariants for the proof of the imperative original version [4] of the algorithm;
- a repository[5] with the complete Dafny source code of the specification, the algorithms and the proofs, and comprehensive documentation;
- some new provably correct simplifications/optimisations;
- some reflections on the practicality of using a verification-aware programming language like Dafny and some lessons learned from this experience.

[3] https://github.com/leonardoalt/ethereum_formal_verification_overview.
[4] https://github.com/dafny-lang/dafny
[5] https://github.com/ConsenSys/deposit-sc-dafny

2 Incremental Merkle Trees

Merkle Trees. A *complete (or perfect) binary tree* is such that each non-leaf node has exactly two children, and the two children have the same *height*. An example of a complete binary tree is given in Fig. 1. A *Merkle (or hash) tree* is a complete binary tree the nodes of which are decorated with *hashes* (fixed-size bit-vectors). The hash values of the leaves are given and the hash values of the internal (non-leaf) nodes are computed by *combining* the values of their children with a binary function **hash**. It follows that a Merkle tree is a complete binary tree decorated with a *synthesised attribute* defined by a binary function.

Merkle trees are often used in distributed ledger systems to define a *property* of a collection of elements e.g., a list L of values. This property can then be used *instead of the collection itself* to verify,[6] using a mechanism called *Merkle proofs*, that data received from a node in the distributed system is not corrupted. This is a crucial optimisation as the size of the collection is usually large (typically up to 2^{32}) and using a compact representation is instrumental to obtain time and space efficient communication and a reasonable transactions' processing throughput.

> *In this work, we are not concerned with Merkle proofs but rather with the (efficient) computation of the* **hash** *attribute on a Merkle tree.*

The actual function used to compute the values of the internal nodes is not relevant in the incremental Merkle tree algorithms' functional logics and without loss of generality we may treat it as a parameter i.e., a given binary function.[7] In the sequel we assume that the decorations of the nodes are integers, and we use in the examples a simple function $\mathbf{hash} : \text{Int} \times \text{Int} \longrightarrow \text{Int}$ defined by $\mathbf{hash}(x, y) = x - y - 1$ instead of an actual (e.g., `sha256`-based) hash function.

Properties of Lists with Merkle Trees. A complete binary tree of height[8] h has 2^h leaves and $2^{h+1} - 1$ nodes. Given a list L of integers (type Int) of size $|L| = 2^h$, we let $T(L)$ be the Merkle tree for L: the values of the leaves of $T(L)$, from left to right, are the elements of L and $T(L)$ is attributed with the **hash** function. The value of the attribute at the root of $T(L)$, the *root hash*, defines a property of the list L. It is straightforward to extend this definition to lists L of size $|L| \leq 2^h$ by right-padding the list with *zeroes* (or any other default values.) Given a list L of size $|L| \leq 2^h$, let $\overline{L}$ denote L right-padded with $2^h - |L|$ default values. The Merkle tree associated with L is $T(\overline{L})$, and the root hash of L is the root hash of $T(\overline{L})$. Computing the root hash of a tree $T(\overline{L})$ requires to traverse all the nodes of the tree and thus is *exponential* in the height of the tree.

The Incremental Merkle Tree Problem. A typical use case of a Merkle tree in the context of Ethereum 2.0 is to represent properties of lists that *grow monotonically*. In the DSC, a Merkle tree is used to record the list of validators

[6] More precisely the verification result holds with high probability as the chosen hashing functions may (rarely) generate collisions.

[7] In the code base, the **hash** function is uninterpreted and its type is generic.

[8] The height is the length of the longest path from the root to any leaf.

and their stakes or deposits. A compact representation of this list, as the root hash of a Merkle tree, is communicated to the nodes in the network rather than the tree (or list) itself. However, as mentioned before, each time a new deposit is appended to the list, computing the new root hash using a standard synthesised-attribute computation algorithm requires exponential time in h. This is clearly impractical in a distributed system like Ethereum in which the height of the tree is 32 and the number of nodes is $2^{33} - 1$.

Given (a tree height) $h > 0$, L a list with $|L| < 2^h$, and e a new element to add to L, the incremental Merkle tree problem (IMTP) is defined as follows:[9]

> *Can we find $\alpha(L)$ a **polynomial-space abstraction** of $T(L)$ such that we can compute in **polynomial-time**: 1) the root hash of $T(L)$ from $\alpha(L)$, and 2) the abstraction $\alpha(L + [e])$ from $\alpha(L)$ and e?*

Linear-time/space algorithms to solve the IMTP were originally proposed by V. Buterin in [4]. However, the correctness of these algorithms is not obvious. In the next section, we analyse the IMTP, and we present the main properties that enable us to *design* polynomial-time recursive algorithms and to *verify* them.

3 Recursive Incremental Merkle Tree Algorithm

In this section we present the main ideas of the *recursive* algorithms to insert a new value in a Merkle tree and to compute the new root hash (after a new value in inserted) by re-using (*dynamic programming*) previously computed results.

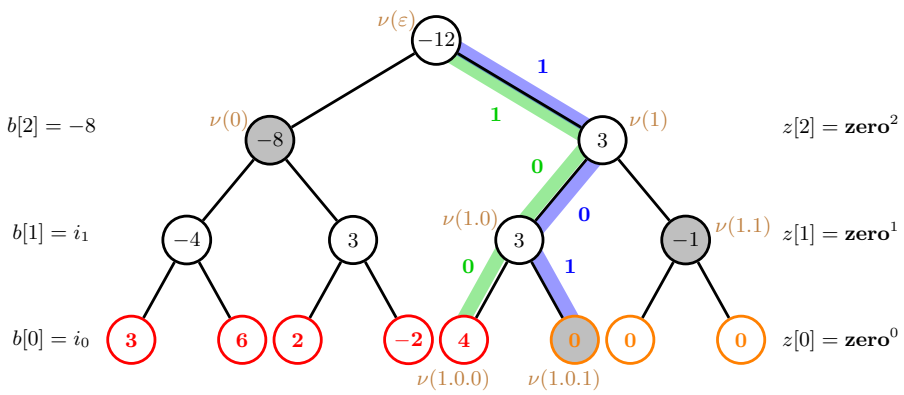

Fig. 1. A Merkle tree of height 3 for list $L_2 = [3, 6, 2, -2, 4]$ and $\mathbf{hash}(x, y) = x - y - 1$. The green path π_1 is encoded as 1.0.0 (from root to leaf) and the blue path π_2 as 1.0.1. The left and right siblings of π_1 are shaded. The values of the right siblings of π_1 at levels 0 and 1 are $z[0] = \mathbf{zero}^0 = 0$ and $z[1] = \mathbf{zero}^1 = \mathbf{hash}(0, 0) = -1$. i_0 and i_1 are arbitrary values.

[9] Polynomial in the height of the tree h. The operator $+$ is list concatenation.

Notations. A *path* π from the root of a tree to a node can be defined as a sequence of bits (left or right) in $\{0,1\}^*$. In a Merkle tree of height h, the *length*, $|\pi|$, of π is at most h. $\nu(\pi)$ is the *node* at the end of π. If $|\pi| = h$ then $\nu(\pi)$ is a leaf. For instance $\nu(\varepsilon)$ is the root of the tree, $\nu(0)$ in Fig. 1 is the node carrying the value -8 and $\nu(1.0.0)$ is a leaf. The *right sibling of a left node* of the form $\nu(\pi.0)$ is the node $\nu(\pi.1)$. Left siblings are defined symmetrically. A node in a Merkle tree is associated with a *level* which is the distance from the node to a leaf in the tree. Leaves are at level 0 and the root is at level h. In a Merkle tree, level 0 has 2^h leaves that can be indexed left to right from 0 to $2^h - 1$. The *n-th leaf* of the tree for $0 \leq n < 2^h$ is the leaf at index n.

Computation of the Root Hash on a Path. We first show that the root hash can be computed if we know the values of the *siblings* of the nodes on *any* path, and the value at the end of the path. For instance, if we know the values of the left and right siblings (shaded nodes in Fig. 1) of the nodes on π_1 (green path in Fig. 1), and the value at the end of π_1, we can compute the root hash of the tree by propagating upwards the attribute **hash**. The value of the **hash** attribute at $\nu(1.0)$ is $\mathbf{hash}(4, \nu(1.0.1)) = 3$, at $\nu(1)$ it is $\mathbf{hash}(3, \nu(1.1)) = 3$ and at the root $\mathbf{hash}(\nu(0), \nu(1)) = \mathbf{hash}(-8, 3) = -12$.

Listing A.1. Recursive Algorithm to Compute the Root Hash.

```
computeRootUp(p:seq<bit>,left:seq<int>,right:seq<int>,seed:int):int
  requires |p| == |left| == |right| //  vectors have the same sizes
  decreases p
{
  if |p| == 0 then seed
  else if last(p) == 0 then // node at end of p is a left node
    computeRootUp(init(p),init(left),init(right),hash(seed,last(right)))
  else // node at end of p is a right node
    computeRootUp(init(p),init(left),init(right),hash(last(left),seed))
}
```

Algorithm `computeRootUp` (Listing A.1) computes[10] bottom-up in time linear in $|\mathtt{p}|$ the root hash with `left` the list of values of the left siblings (top-down) on a path `p` (top-down), `right` the values of the right siblings (top-down) and `seed` the value at $\nu(\mathtt{p})$. The generic version (uninterpreted hash function) of the algorithm is provided in the `ComputeRootPath.dfy` file.

For the green path $\mathtt{pi}_1 = [1,0,0]$ in Fig. 1, $\mathtt{left} = [-8, i_1, i_0]$, $\mathtt{right} = [-1,-1,0]$ and the seed is 4. The evaluation of `computeRootUp` returns -12.

Given a path π, if the leaves on the right of $\nu(\pi)$ all have the default value 0, the values of the right siblings on the path π only depend on the *level* of the sibling in the tree. For example, the leaves on the right of π_1 (orange in Fig. 1) all have the default value 0. The root hash of a tree in which all the leaves have the same default value only depends on the level of the root: 0 at level 0, $\mathbf{hash}(0,0)$ at level 1, $\mathbf{hash}(\mathbf{hash}(0,0), \mathbf{hash}(0,0))$ at level 2 and so on. Let $\mathbf{zero}^l$ be defined by: $\mathbf{zero}^l = 0$ if $l = 0$ else $\mathbf{hash}(\mathbf{zero}_0^{l-1}, \mathbf{zero}_0^{l-1})$.

[10] For $l = l' + x$, $\mathtt{last}(l) = x$, $\mathtt{init}(l) = l'$, and for $l = x + l'$, $\mathtt{first}(l) = x$, $\mathtt{tail}(l) = l'$.

Given a path π, if all the leaves on the right of $\nu(\pi)$ have the default value, any right sibling value at level l on π is equal to $\mathbf{zero}^l$.

As an example in Fig. 1, the right siblings on $\pi_1 = 1.0.0$ have values 0 at level 0, node $\nu(1.0.1)$, and $\mathbf{hash}(0,0) = \mathbf{zero}^1 = -1$ at level 1, node $\nu(1.1)$. If a path `p` leads to a node with the default value 0 and all the leaves right of $\nu(\mathtt{p})$ have the default value 0, the root hash depends only on the values of the `left` and default `right` siblings. Hence the root hash can be obtained by `computeRootUp(p, left, right, 0)`. For the path `pi2` $= [1,0,1]$ (Fig. 1), `left` $= [-8, i_1, 4]$ and `right` $= [-1,-1,0]$, `computeRootUp(pi2, left, right, 0)` returns -12.

As a result, to compute the root hash of a tree $T(\overline{L})$, we can use a compact abstraction $\alpha(L)$ of $T(\overline{L})$ composed of the left siblings vector b and the right siblings default values z (Fig. 1) of the path to the $|L|$-th leaf in $T(\overline{L})$.

Insertion: Update the Left Siblings. Assume π_1 is a path to the n-th leaf and $n < 2^h - 1$ (not the last leaf), where the next value v is to be inserted. As we have shown before, if we have b_1 holding the values of left siblings of π_1, z and v, we can compute the new attribute values of the nodes on π_1 and the new root hash after v is inserted. Let π_2 be the path to the $n+1$-th leaf. If we can compute the values b_2 of the left siblings of π_2 as a function of b_1, z and v, we have an efficient algorithm to *incrementally* compute the root hash of a Merkle tree: we keep track of the values of the left siblings b on the path to the next available leaf, and iterate this process each time a new value is inserted.

As $\nu(\pi_1)$ is not the last leaf, π_1 must contain at least one 0, and has the form[11] $\pi_1 = w.0.1^k$ with $w \in \{0,1\}^*, k \geq 0$. Hence, the path π_2 to the $n+1$-th leaf is $w.1.0^k$, arithmetically $\pi_2 = \pi_1 + 1$. An example of two consecutive paths is given in Fig. 1 with π_1 (green) and π_2 (blue) to the leaves at indices 4 and 5.

The related forms of π_1 (a path) and π_2 (the successor path) are useful to figure out how to incrementally compute the left siblings vector b_2 for π_2:

- as the initial prefix w is the same in π_1 and π_2, the values of the left siblings on the nodes of w are the same in b_1 and b_2;
- all the nodes in the suffix 0^k of π_2 are left nodes and have right siblings. It follows that the corresponding k values in b_2 are irrelevant as they correspond to right siblings, and we can re-use the corresponding b_1 values;
- hence b_2 is equal to b_1 except possibly for the level of the node at $\nu(w.0)$.

We now illustrate how to compute the new value in the vector b_2 on the example of Fig. 1. Let $\pi_1 = w.0$ and $\pi_2 = w.1$ with $w = 1.0$ and $|w| = 2$. For the top levels 2 and 1, b_2 is the same as b_1: $b_2[2] = b_1[2] = -8$ and $b_2[1] = b_1[1] = i_1$. For level 0, the level of the node $\nu(w.0)$, the value at $\nu(w.0) = \nu(1.0.0)$ becomes the left sibling of the node $\nu(1.0.1)$ on π_2 at this level. So the new value of the left sibling on π_2 is exactly the new value, 4, of the node $\nu(1.0.0)$ after 4 is inserted.

More generally, when computing the new root hash bottom-up on π_1, the first time we encounter a left node, at level d, we update the corresponding value of b with the computed value of the attribute on π_1 at level d. Algorithm[12]

[11] $x^k, x \in \{0,1\}$ denotes the sequence of k x's.

[12] $+$ stands for list concatenation.

`insertValue` in Listing A.2 computes, in linear-time, the list of values of the left siblings (top-down) of the path $\mathtt{p}+1$ using as input the list (top-down) of values left (resp. right) siblings `left` (resp. `right`) of `p` and `seed` the new value inserted at $\nu(\mathtt{p})$. The generic (non-interpreted hash) algorithm is provided in the `NextPathInCompleteTreesLemmas.dfy` file.

Listing A.2. Recursive Algorithm to Compute the New Left Siblings.

```
insertValue(p:seq<bit>,left:seq<int>,right:seq<int>,seed:int):seq<int>
  requires |left| == |right| == |p| >= 1
  decreases p
{
  if |p| == 1 then  // note that first(p) == last(p) in this case
    if first(p) == 0 then [seed] else left
  else if last(p) == 0 then // we encounter a left node. Stop recursion.
      init(left) + [seed]
  else //  right node,move up on the path.
      insertValue(init(p),init(left),init(right),hash(last(left),seed))
        + [last(left)]
}
```

We illustrate how the algorithm `insertValue` works with the example of Fig. 1. Assume we insert the seed 4 at the end of the (green) path `pi1 = [1,0,0]`. The left (resp. right) siblings' values are given by `left` $= [-8, i_1, i_0]$ (resp. `right` $= [-1, -1, 0]$). `insertValue` computes the values of the left siblings on the (blue) path `pi2 = [1,0,1]` after 4 is inserted at the end of π_1: the first call terminates the algorithm and returns $[-8, i_1, 4]$ which is the list of left siblings that are needed on π_2.

In the next section we describe how to verify the recursive algorithms and the versions implemented in the DSC.

4 Verification of the Algorithms

In order to verify the implemented (imperative style/Solidity) versions of the algorithms of the DSC, we first prove total correctness of the recursive versions (Section 3) and them use them to prove the code implemented in the DSC.

> *In this section, the Dafny code has been simplified and sometimes even altered while retaining the main features, for the sake of clarity. The code in this section may not compile. We provide the links to the files with the full code in the text and refer the reader to those files.*

Correctness Specification. The (partial) correctness of our algorithms reduces to checking that they compute the same values as the ones obtained with a synthesised attribute on a Merkle tree. We have specified the data types `Tree`, `MerkleTree` and `CompleteTrees` and the relation between Merkle trees and lists of values. The defintions are provided in the `trees` folder.

The root hash of a `MerkleTree t` is `t.rootv`. The (specification) function `buildMerkle(h, L, hash)` returns a `MerkleTree` of height `h`, the leaves of

which are given by the values (right-padded) $\overline{\mathtt{L}}$, and the values on the internal nodes agree with the definition of the synthesised attribute **hash**, i.e., what we previously defined in Section 2 as $T(\overline{\mathtt{L}})$. It follows that `buildMerkle(h, L, hash).rootv` is the value of the root hash of a Merkle tree with leaves $\overline{\mathtt{L}}$.

Total Correctness. The total correctness proof for the `computeRootUp` function amounts to showing that 1) the algorithm always terminates and 2) the result of the computation is the same as the hash of the root of the tree. In Dafny, to prove termination, we need to provide a ranking function (strictly decreasing and bounded from below.) The length of the path `p` is a suitable ranking function (see the `decreases` clause in Listing A.1) and is enough for Dafny to prove termination of `computeRootUp`.

We establish property 2) by proving a *lemma* (Listing A.3): the pre-conditions (`requires`) of the lemma are the assumptions, and the post-conditions (`ensures`) the intended property. The body of the lemma (with a non-interpreted hash function) which provides the machine-checkable proof is available in the computeRootPath.dfy file.

Listing A.3. Correctness Proof Specification for `ComputeRootUp`.

```
lemma computeRootUpIsCorrectForTree(
  p:seq<bit>,r:Tree<int>,left:seq<int>,right:seq<int>,seed:int)
  //  size of p is the height of the tree r
  requires |p| == height(r)
  //  r is a Merkle tree for attribute hash
  requires isCompleteTree(r)
  requires isDecoratedWith(hash,r)
  //  the value at the end of the path p in r is seed
  requires seed == nodeAt(p,r).v
  //  vectors of same sizes
  requires |right| == |left| == |p|
  // Left and right contain values of left and right siblings of p in r.
  requires forall i :: 0 <= i < |p| ==>
      //  the value of the sibling of the node at p[..i] in r
      siblingValueAt(p,r,i + 1) ==
      //  are stored in left and right
      if p[i] == 0 then right[i] else left[i]
  //  Main property: computeRootUp computes the hash of the root of r
  ensures r.rootv == computeRootUp(p,left,right,seed)
```

Lemma `computeRootUpIsCorrectForTree` requires that the tree `r` is a Merkle tree, and that the lists `left` (resp. `right`) store the values of left (resp. right) siblings of the nodes on a path `p`. Moreover, the value at the end of `p` should be `seed`. Under these assumptions the conclusion (`ensures`) is that `computeRootUp` returns the value of the root hash of `r`.

The proof of lemma `computeRootUpIsCorrectForTree` requires a few intermediate sub-lemmas of moderate difficulty. The main step in the proof is to establish an equivalence between a bottom-up computation `computeRootUp` and the top-down definition of (attributed) Merkle trees. All the proofs are by induction on the tree or the path. The complete Dafny code for this algorithm is available in computeRootPath.dfy file.

Termination for `insertValue` is proved by using a ranking function (decreases clause in Listing A.2). The functional correctness of `insertValue` reduces to proving that, assuming `left` (resp. `right`) contains the values of the left (resp. right) siblings of the nodes on `p`, then `insertValue(p, left, right, seed)` returns the values of the nodes that are left siblings on the successor path. The code for this lemma is in the NextPathInCompleteTreesLemmas.dfy file. The main proof is based on several sub-lemmas that are not hard conceptually but cannot be easily discharged using solely the built-in Dafny induction strategies. They require some intermediate proof hints (verified calculations) to deal with all the nodes on the path `p`.

Listing A.4. `ComputeRootUpWithIndex`.

```
computeRootUpWithIndex(
  h:nat,k:nat,left:seq<int>,right:seq<int>,seed:int):int
  requires |left| == |right| == h
  //  the index is in the range of indices for a tree of height h
  requires k < power2(h)
  // Indexed algorithm computes the same value as computeRootUp
  ensures computeRootUpWithIndex(h,k,left,right,f,seed) ==
      //  natToBitList(k,h) is the binary encoding of k over h bits
      computeRootUp(natToBitList(k,h),left,right,f,seed)
  //  ranking function
  decreases h
{
  if h == 0 then seed
  else if k % 2 == 0 then //  left node
  computeRootUpWithIndex(h-1,k/2,init(left),init(right),hash(seed,last(right)))
  else // right node
  computeRootUpWithIndex(h-1,k/2,init(left),init(right),hash(last(left),seed))
}
```

Index Based Algorithms. The algorithms that implement the DSC do not use a bitvector to encode a path, but rather, a *counter* that records the number of values inserted so far and the height of the tree. In order to prove the algorithms actually implemented in the DSC, we first recast the `computeRootUp` and `insertValue` algorithms to use a counter and the height h of a tree. In this step, we use a parameter `k` that is the index of the next available leaf where a new value can be inserted. The leaves are indexed left to right from 0 to $2^h - 1$ and hence k is the number of values that have been inserted so far. It follows that the leaves with indices $k \leq i \leq 2^h - 1$ have the default value. The correspondence between the bitvector encoding of the path to the leaf at index k and the value k is straightforward: the encoding of the path p is the value of k in binary over h bits. We can rewrite left `computeRootUp` to use use k and h (`computeRootUpWithIndex`, Listing A.4) and prove it computes the same value as `computeRootUp`. A similar proof can be established for the `insertValue` algorithm. The index based algorithms and the proofs that they are equivalent (compute the same values as) to `computeRootUp` and `insertValue` are available in the IndexBasedAlgorithm.dfy file. Dafny can discharge the equivalence proofs with minimal proof hints using the builtin induction strategies.

Total Correctness of the Algorithms Implemented in the DSC. In this section we present the final proof of (total) correctness for the algorithms implemented in the DSC (Solidity-like version.) Our proof establishes that the imperative versions, with while loops and dynamic memory allocation (for arrays) are correct, terminate and are memory safe.

The DSC is an object and has a state defined by a few variables: `count` is the number of inserted values (initially zero), `branch` is a vector that stores that value of the left siblings of the path leading to the leaf at index `count`, and `zero_hashes` is what we previously defined as z. The algorithm that computes the root hash of the Merkle tree in the DSC is `get_deposit_root()`. `get_deposit_root()` does not have any *seed* parameter as it computes the root hash using the default value (0). The correctness proof of `get_deposit_root()` uses the functional (proved correct) algorithm `computeRootUpWithIndex` as an invariant. Listing A.5 is a simplified version (for clarity) of the full code available in the DepositSmart.dfy file.

Listing A.5. Implemented Version of `computeRootUp`.

```
method get_deposit_root() returns (r:int)
  //  The result of get_deposit_root_() is the root value of the Merkle tree
  //  values is a ghost variable and records all the inserted values
  ensures r == buildMerkle(values,TREE_HEIGHT,hash).rootv
{
  //  Store the expected result in a ghost variable.
  ghost var e :=computeRootUpWithIndex(TREE_HEIGHT,count,branch,zero_hashes,0);
  //  Start with default value for r.
  r := 0;
  var h := 0;
  var size := count;
  while h < TREE_HEIGHT
    //  Main invariant:
    invariant e == computeRootUpWithIndex(
          TREE_HEIGHT - h,size,
          take(branch,TREE_HEIGHT - h),take(zero_hashes,TREE_HEIGHT - h),r)
  {
    if size % 2 == 1 {
      r := hash(branch[h],r);
    } else {
      r := hash(r,zero_hashes[h]);
    }
    size := size / 2;
    h := h + 1;
  }
}
```

The algorithm that inserts a value `v` in the tree is `deposit(v)` in the implemented version of the DSC. Listing A.6 is an optimised version of the original algorithm. The simplification is explained in Section 5. The correctness of the algorithm is defined by ensuring that, if at the beginning of the computation the vectors `branch` (resp, `zero_hashes`) contain values of the left (resp. right) siblings of the path leading to the leaf at index `count`, then at the end of the computation, after `v` is inserted, this property still holds. The proof of this invari-

ant requires a number of proof hints for Dafny to verify it. We use the functional version of the algorithm to specify a loop invariant (not provided in Listing A.6).

The termination proof is easy using `size` as the decreasing ranking function. However, a difficulty in this proof is memory safety, i.e. to guarantee that the index `i` used to access `branch[i]` is within the range of the indices of `branch`.

We have also proved the initialisation functions `init_zero_hashes()` and the DSC `constructor`. The full code of the imperative version of the DSC is available in the DepositSmart.dfy file.

Listing A.6. The `deposit` method.

```
method deposit(v:int)
  //  The tree cannot be full
  requires count < power2(TREE_HEIGHT) - 1
  //  branch and zero_hashes hold the values of the siblings
  requires areSiblingsAtIndex(|values|,
    buildMerkle(values,TREE_HEIGHT,hash),branch, zero_hashes)
  //  Correctness property
  ensures areSiblingsAtIndex(|values|,
    buildMerkle(values,TREE_HEIGHT,hash),branch,zero_hashes)
{
  var value := v;
  var size : nat := count;
  var i : nat := 0;
  //  Store the expected result in e.
  ghost var e := computeLeftSiblingsOnNextpathWithIndex(
    TREE_HEIGHT,old(size),old(branch),zero_hashes,v);
  while size % 2 == 1
      //  Main invariant:
      invariant e ==
      computeLeftSiblingsOnNextpathWithIndex(
          TREE_HEIGHT - i,size,
          take(branch,TREE_HEIGHT - i),
          take(zero_h,TREE_HEIGHT - i),value) + drop(branch,TREE_HEIGHT - i)
      decreases size
  {
      value := f(branch[i],value);
      size := size / 2;
      i := i + 1;
  }
  //  0 <= i < |branch| and no there is no index-out-of-bounds error
  branch[i] := value;
  count := count + 1;
  values := values + [v];
}
```

5 Findings and Lessons Learned

Methodology. In contrast to the previous attempts to analyse the DSC, we have adopted a textbook approach and used standard algorithms' design techniques (e.g., dynamic programming, refinement, recursion.) This has several advantages over a direct proof (e.g., [20]) of the imperative code including:

- the design of simple algorithms and proofs;
- recursive and language-agnostic recursive versions of the algorithms;

– new and provably correct simplifications/optimisations.

Algorithmic Considerations. Our implementations and formal proofs have resulted in the identification of two previously unknown/unconfirmed optimisations. First, it is not necessary to initialise the vector of left siblings, `b`, and the algorithms are correct for any initial value of this vector.

Second, the original version of the `deposit` algorithm (which we have proved correct too) has the form[13] given in Listing A.7. Our formal machine-checkable proof revealed[14] that indeed the condition `C1` is always true and the loop always terminates because `C2` eventually becomes true. As witnessed by the comment after the loop in the Solidity code of the DSC, this property was expected but not confirmed and the Solidity contract authors did not take the risk to simplify the code. Our result shows that the algorithm can be simplified to `while not(C2) do ... od`.

Listing A.7. Solidity Version of the DSC Deposit Function.

```
deposit( ... )
{
  while C1 do
    if C2 return;
  ...
  od
  //  As the loop should always end prematurely with the `return` statement,
  //  this code should be unreachable. We assert `false` just to be safe.
  assert(false);
}
```

This is interesting not only from a safety and algorithmic perspectives, but also because it reduces the computation cost (in gas/Ether) of executing the `deposit` method. This simplification proposal is currently being discussed with the DSC developer, however the currently deployed version still uses the non-optimised code.

Verification Effort. The verification effort for this project is 12 person-weeks resulting in 3500 lines of code and 1000 lines of documentation. This assumes familiarity with program verification, Hoare logic and Dafny. Table 1, page 15 provides some insights into the code base. Note that in Table 1 we do not report on the verification time (e.g. time spent in Z3) as it is known to be an irrelevant metrics in Dafny/Z3: implemented optimisations may use clever cashing hash tables and even a name change can drastically impact the verification. What Dafny/Z3 guarantees is that the *result* of the verification is the sound but not that the verification time is stable.

The filenames in green are the ones that require the less number of hints for Dafny to check a proof. In this set of files the hints mostly consist of simple *verified calculations* (e.g., empty sequence is a neutral element for lists `[] + l == l + [] == l`.) Most of the results on sequences (`helpers` package) and

[13] The complete Solidity source code is freely available on GitHub at https://github.com/ethereum/eth2.0-specs/blob/dev/solidity_deposit_contract/deposit_contract.sol

[14] This finding was not uncovered in any of the previous audits/analyses.

simplifications of sequences of bits (`seqofbits` package) are in this category and require very few hints. This also applies for the proofs[15] of the `algorithms` package, e.g., proving that the versions using the index of a leaf instead of the binary encoding of a path are equivalent.

The filenames in orange require some non-trivial proof hints beyond the implicit induction strategies built in Dafny. For instance in NextPathInCompleteTrees.dfy and PathInCompleteTrees.dfy, we had to provide several annotations and structure for the proofs. This is due to the fact that the proofs involve properties on a Merkle tree t_1 and its *successor* t_2 (after a value is inserted) which is a new tree, and on a path π_1 in t_1 and its successor π_2 in t_2.

The filenames in red require a lot of hints. For the files in the `synthattribute` package it is mostly calculation steps. Some steps are not absolutely necessary but adding them reduces the verification time by on order of magnitude (on our system configuration, MacBookPro 16GBRAM). The hardest proof is probably the correctness of the `deposit` method in DepositSmart.dfy. The proof requires non trivial lemmas and invariants. The difficulty stems from a combination of factors: first the while loop of the algorithm (Listing A.6) maintains a constraint between `size` and `i`, the latter being used to access the array elements in `branch`. Proving that there is no array-of-bounds error (i.e., i is within the size of `branch`) requires to prove some arithmetic properties. Second, the proof of the main invariant (Listing A.6) using the functional specification `computeLeftSiblingsOnNextpathWithIndex` is complex and had to be structured around additional lemmas.

Overall, almost 90% of the lines of code are (non-executable) proofs, and function definitions used in the proofs. The verified algorithms implemented in the DSC functional are provided in `DepositSmart.dfy` and account for less than 10% of the code.

Considering the criticality of the DSC (it processes millions of ETH), 12 person-weeks can be considered a moderate effort well worth the investment: the result is an unparalleled level of trustworthiness that can inspire confidence in the Ethereum platform. According to our experts (ConsenSys Diligence) in the verification of Smart Contracts, the size of such an effort is realistic and practical considering the level of guarantees provided. The only downside is the level of verification expertise required to design the proofs.

The trust base in our work is composed of the Dafny verification engine (verification conditions generator) and the SMT-solver Z3.

Dafny Experience. Dafny has excellent documentation, support for data structures, functional (side-effect free) and object-oriented programming. The automated verification engine has a lot of built-in strategies (e.g., induction, calculations) and a good number of specifications are proved fully automatically without providing any hints. The Dafny proof strategies and constructs that we

[15] The file `CommuteProof.dfy` in this package is not needed for the main proof but was originally used and provides an interesting result, so it is still in code base.

mostly used are *verified calculations* and *induction*. The side-effect free proofs seem to be handled much more efficiently (time-wise) than the proofs using mutable data structures.

src/dafny/**package**/file.dfy	#LoC	Lemmas	Methods	#Doc	(#Doc/#LoC in %)
smart					
DepositSmart.dfy	163	1 + 1	1 + 3	92	56
smart/algorithms					
CommuteProof.dfy	73	2	0	31	42
IndexBasedAlgorithm.dfy	96	3	2	59	61
MainAlgorithm.dfy	66	2	0	38	58
OptimalAlgorithm.dfy	24	2	0	15	62
Sub-total	**259**	**2 + 7**	**2**	**143**	**55**
smart/helpers					
Helpers.dfy	51	5	1	10	20
SeqHelpers.dfy	137	10	6	34	25
smart/paths					
NextPathInCompleteTrees.dfy	262	1 + 2	2	99	38
PathInCompleteTrees.dfy	408	2 + 13	0	60	15
Sub-total	**670**	**3 + 15**	**2**	**159**	**24**
smart/seqofbits					
SeqOfBits.dfy	527	19	0	100	19
smart/synthattribute					
ComputeRootPath.dfy	305	2 + 9	0	116	38
GenericComputation.dfy	148	6	0	75	51
RightSiblings.dfy	210	1 + 2 + 2	1	57	27
Siblings.dfy	124	1 + 1	0	31	25
SiblingsPlus.dfy	556	2 + 2	0	52	9
Sub-total	**1343**	**4 + 4 + 20**	**1**	**331**	**25**
smart/trees					
CompleteTrees.dfy	89	8	1	19	21
MerkleTrees.dfy	208	6	3	101	49
Trees.dfy	91	3	5	41	45
Sub-total	**388**	**17**	**9**	**161**	**41**
src/dafny					
TOTAL	**3538**	**5 + 9 + 94**	**1 + 24**	**1030**	**29**

Table 1. Dafny Code Statistics. #Loc (resp. #Doc) is the number of Lines of Code (resp. Documentation), Lemmas is the number of proofs broken down in difficulty levels, Methods the number of executable methods/function methods. Colour scheme easy/few proof hints, moderate, hard/detailed proof hints.

In the current version we have used the `autocontracts` attribute for the DSC object which is a convenient way of proving memory safety using a specific invariant (given by the `Valid` predicate). This could probably be optimised as Dafny has some support to specify precisely the side-effects using *frames* (based on *dynamic framing*.)

Overall, Dafny is a practical option for the verification of mission-critical smart contracts, and a possible avenue for adoption could be to extend the Dafny code generator engine to support Solidity, a popular language for writing smart contracts for the Ethereum network, or to automatically translate Solidity

into Dafny. We are currently evaluating these options with our colleagues at ConsenSys Diligence, as well as the benefits of our technique to the analysis of other critical smart contracts.

The software artefacts including the implementations, proofs, documentation and a Docker container to reproduce the results are freely available as a GitHub repository at https://github.com/ConsenSys/deposit-sc-dafny.

Acknowledgements. I wish to thank Suhabe Bugrara, ConsenSys Mesh, for helpful discussions on the DSC's previous work and the anonymous reviewers of a preliminary version of this paper.

References

1. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In: Andronick, J., Felty, A.P. (eds.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 66–77. ACM (2018), https://doi.org/10.1145/3167084
2. Andrici, C., Ciobâcă, Ş.: Verifying the DPLL algorithm in Dafny. In: Marin, M., Craciun, A. (eds.) Proceedings Third Symposium on Working Formal Methods, FROM 2019, Timişoara, Romania, 3-5 September 2019. EPTCS, vol. 303, pp. 3–15 (2019), https://doi.org/10.4204/EPTCS.303.1
3. Bugrara, S.: A review of the deposit contract (2020), https://github.com/suhabe/eth-deposit-contract-vyper-review/blob/master/EthDepositContractVyperReview.pdf
4. Buterin, V.: Progressive merkle tree, https://github.com/ethereum/research/blob/master/beacon_chain_impl/progressive_merkle_tree.py
5. Chen, X., Rosu, G.: $\mathbb{K}$ - A semantic framework for programming languages and formal analysis. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) Engineering Trustworthy Software Systems - 5th International School, SETSS 2019, Chongqing, China, April 21-27, 2019, Tutorial Lectures. Lecture Notes in Computer Science, vol. 12154, pp. 122–158. Springer (2019), https://doi.org/10.1007/978-3-030-55089-9_4
6. ConsenSys Diligence: Mythx, https://mythx.io/
7. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 273–289. Springer International Publishing, Cham (2015)
8. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12031, pp. 161–179. Springer (2019), https://doi.org/10.1007/978-3-030-41600-3_11
9. Hajdu, Á., Jovanovic, D., Ciocarlie, G.F.: Formal specification and verification of solidity contracts with events (short paper). In: Bernardo, B., Marmsoler, D. (eds.) 2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020, July 20-21, 2020, Los Angeles, California, USA (Virtual Conference). OASIcs, vol. 84, pp. 2:1–2:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020), https://doi.org/10.4230/OASIcs.FMBC.2020.2

10. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: Miller, E.L., Hand, S. (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015. pp. 1–17. ACM (2015), https://doi.org/10.1145/2815400.2815428
11. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009. pp. 207–220. ACM (2009), https://doi.org/10.1145/1629575.1629596
12. Lammich, P.: Efficient verified implementation of introsort and pdqsort. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12167, pp. 307–323. Springer (2020), https://doi.org/10.1007/978-3-030-51054-1_18
13. Leino, K.R.M.: Accessible software verification with Dafny. IEEE Softw. 34(6), 94–97 (2017), https://doi.org/10.1109/MS.2017.4121212
14. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. 43(4), 363–446 (2009), https://doi.org/10.1007/s10817-009-9155-4
15. Mai, H., Pek, E., Xue, H., King, S.T., Madhusudan, P.: Verifying security invariants in expressos. In: Sarkar, V., Bodík, R. (eds.) Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013. pp. 293–304. ACM (2013), https://doi.org/10.1145/2451116.2451148
16. Matias, M.: Program Verification of FreeRTOS Using Microsoft Dafny. Cleveland State University (2014), https://books.google.com.au/books?id=A_iyoQEACAAJ
17. de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean Theorem Prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9195, pp. 378–388. Springer (2015), https://doi.org/10.1007/978-3-319-21401-6_26
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
19. Ogawa, M., Horita, E., Ono, S.: Proving properties of incremental merkle trees. In: Nieuwenhuis, R. (ed.) Automated Deduction – CADE-20. pp. 424–440. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
20. Park, D., Zhang, Y.: Formal verification of the incremental merkle tree algorithm (2020), https://github.com/runtimeverification/verified-smart-contracts/blob/master/deposit/formal-incremental-merkle-tree-algorithm.pdf
21. Park, D., Zhang, Y., Rosu, G.: End-to-end formal verification of ethereum 2.0 deposit smart contract. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp. 151–164. Springer (2020), https://doi.org/10.1007/978-3-030-53288-8_8
22. Paulin-Mohring, C.: Introduction to the Coq Proof-Assistant for Practical Software Verification, pp. 45–95. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-35746-6_3
23. Pearce, D.J., Utting, M., Groves, L.: An introduction to software verification with Whiley. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) Engineering Trustworthy Software

Systems - 4th International School, SETSS 2018, Chongqing, China, April 7-12, 2018, Tutorial Lectures. Lecture Notes in Computer Science, vol. 11430, pp. 1–37. Springer (2018), https://doi.org/10.1007/978-3-030-17601-3_1
24. Peña, R.: An assertional proof of red-black trees using dafny. J. Autom. Reason. 64(4), 767–791 (2020), https://doi.org/10.1007/s10817-019-09534-y
25. Runtime Verification Inc.: Formally verified smart contracts., https://github.com/runtimeverification/verified-smart-contracts
26. Sternagel, C.: Proof pearl–a mechanized proof of ghc's mergesort. J. Autom. Reason. 51(4), 357–370 (Dec 2013), https://doi.org/10.1007/s10817-012-9260-7
27. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 61–78. Springer International Publishing, Cham (2018)
28. Wüstholz, V., Christakis, M.: Harvey: A Greybox Fuzzer for Smart Contracts, pp. 1398–1409. Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3368089.3417064