Formally Verified EVM Block-Optimizations*

Elvira Albert[1], Samir Genaim[1], Daniel Kirchner[2], and Enrique Martín-Martín[1]

[1] Complutense University of Madrid, Spain
[2] Ethereum Foundation

Abstract. The efficiency and the security of *smart contracts* are their two fundamental properties, but might come at odds: the use of optimizers to enhance efficiency may introduce bugs and compromise security. Our focus is on `EVM` (Ethereum Virtual Machine) *block-optimizations*, which enhance the efficiency of jump-free blocks of opcodes by eliminating, reordering and even changing the original opcodes. We reconcile efficiency and security by providing the verification technology to formally prove the correctness of `EVM` block-optimizations on smart contracts using the Coq proof assistant. This amounts to the challenging problem of proving semantic equivalence of two blocks of `EVM` instructions, which is realized by means of three novel Coq components: a symbolic execution engine which can execute an `EVM` block and produce a symbolic state; a number of simplification lemmas which transform a symbolic state into an equivalent one; and a checker of symbolic states to compare the symbolic states produced for the two `EVM` blocks under comparison.

1 Introduction

In many contexts, security requirements are critical and formal verification today plays an essential role to verify/certify these requirements. One of such contexts is the blockchain, in which software bugs on *smart contracts* have already caused several high profile attacks (e.g., [16,15,28,35,14,13]). There is hence huge interest and investment in guaranteeing their correctness, e.g., Certora [1], Veridise [2], apriorit [3], Consensys [4], Dedaub [5] are companies that offer smart contract audits using formal methods' technology. In this context, efficiency is of high relevance as well, as deploying and executing smart contracts has a cost (in the corresponding cryptocurrency). Hence, optimization tools for smart contracts have emerged in the last few years (e.g., `ebso` [27], `SYRUP` [11], `GASOL` [10], the `solc` optimizer [9]). Unfortunately, there is a dichotomy of efficiency and correctness: as optimizers can be rather complex tools (not formally verified), they might introduce bugs and potential users might be reluctant of optimizing their code. This has a number of disruptive consequences: owners will pay more to deploy (non-optimized) smart contracts; clients will pay more to run transactions every time they are executed; the blockchain will accept less transactions as they are more costly. Rather than accepting such a dichotomy, our work tries to

* This work is funded by the Ethereum Foundation under Grant ID FY22-0698.

overturn it by developing a fully automated formal verification tool for proving the correctness of the optimized code.

The general problem addressed by the paper is formally verifying semantic equivalence of two bytecode programs, an initial code I and an optimization of it O –what is considered a great challenge in formal verification. For our purpose, we will narrow down the problem by (1) considering fragments of code that are *jump-free* (i.e., they do not have loops nor branching), and by (2) considering only stack EVM operations (memory/storage opcodes and other blockchain-specific opcodes are not considered). These assumptions are realistic as working on jump-free blocks still allows proving correctness for optimizers that work at the level of the blocks of the CFG (e.g., super-optimizers [27,11,10] and many rule-based optimizations performed by the Solidity compiler [9]). Considering only stack optimizations, and leaving out memory and storage simplifications and blockchain-specific bytecodes, does not restrict the considered programs, as we work at the smaller block partitions induced by the not handled operations found in the block (splitting into the block before and after). Even in our narrowed setting, the problem is challenging as block-optimizations can include any elimination, reorder and even change of the original bytecodes.

Consider the next block I, taken from a real smart contract [8]. The GASOL optimizer [10], relying on the commutativity of OR and AND, optimizes it to O:

```
I: PUSH2 0x100 PUSH1 0x1 PUSH1 0xa8 SHL SUB NOT SWAP1 SWAP2 AND PUSH1 0x8 SWAP2 SWAP1
   SWAP2 SHL PUSH2 0x100 PUSH1 0x1 PUSH1 0xa8 SHL SUB AND OR PUSH1 0x5
O: PUSH2 0x100 PUSH1 0x1 PUSH1 0xa8 SHL SUB DUP1 NOT SWAP2 PUSH1 0x8 SHL AND
   SWAP2 AND OR PUSH1 0x5
```

This saves 11 bytes because (1) the expression SUB(SHL(168,1),256) –that corresponds to "PUSH2 0x100 PUSH1 0x1 PUSH1 0xa8 SHL SUB" – is computed twice; but it can be duplicated if the stack operations are properly made saving 8 bytes; and (2) two SWAPs are needed instead of 5, saving 3 more bytes.

This paper proposes a technique, and a corresponding tool, to automatically verify the correctness of EVM block-optimizations (as those above) on smart contracts using the Coq proof assistant. This amounts to the challenging problem of proving semantic equivalence of two blocks of EVM instructions, which is realized by means of three main components which constitute our main contributions (all formalized and proven correct in Coq): (1) a symbolic interpreter in Coq to symbolically execute the EVM blocks I and O and produce resulting symbolic states S_I and S_O, (2) a series of simplification rules, which transform S_I and S_O into equivalent ones S'_I and S'_O, (3) a checker of symbolic states in Coq to decide if two symbolic states S'_I and S'_O are semantically equivalent.

2 Background

The Ethereum VM (EVM) [36] is a stack-based VM with a word size of 256-bits that is used to run the smart contracts on the Ethereum blockchain. The EVM has the following categories of bytecodes: (*1*) Stack operations; (*2*) Arithmetic operations; (*3*) Comparison and bitwise logic operations;(*4*) Memory and storage manipulation;(*5*) Control flow operations; (*6*) Blockchain-specific opcodes, e.g., block and transaction environment information, compute hash, calls, etc. The first three types of opcodes are handled within our verifier, and handling optimizations on opcodes of types 4-6 is discussed in Sec. 6.

The focus of our work is on optimizers that perform optimizations only at the level of the blocks of the CFG (i.e., intra-block optimizations). A well-known example is the technique called *super-optimization* [24] which, given a loop-free sequence of instructions searches for the optimal sequence of instructions that is semantically equivalent to the original one and has optimal cost (for the considered criteria). This technique dates back to 1987 and has had a revival [29,23] thanks to the availability of SMT solvers that are able to do the search efficiently. We distinguish two types of possible intra-block optimizations: (*i*) Rule-based optimizations which consist in applying arithmetic/bitwise simplifications (see Fig. 1); and (*ii*) Stack-data optimizations which consist in searching for alternative stack operations that lead to an output stack with exactly the same data.

The first type of optimizations are applied by the optimizer integrated in the Solidity compiler [9] as rule transformations, and they are also applied by `EVM` optimizers in different ways. `ebso` [27] encodes the semantics of arithmetic and bitwise operations in the SMT encoding so that the SMT solver searches for these optimizations together with those of type (ii). Instead, `SYRUP` [11] and `GASOL` [10] apply rule-based optimizations in a pre-phase and leave to the SMT solver only the search for the second type of optimizations. This classification of optimizations is also relevant for our approach as (i) will require integrating and proving all simplification rules correct (Sec. 4.2) while (ii) are implicit within the symbolic execution (Sec. 4.1). A block of `EVM` code that has been subject to optimizations of the two types above is in principle "provable" using our tool.

There is not much work yet on formalizing the `EVM` semantics in Coq. One of the most developed approaches is [20], which is a definition of the `EVM` semantics in the Lem [26] language that can be exported to interactive theorem provers like Isabelle/HOL or Coq. According to the comparison in [19], this implementation of `EVM` "is executable and passes all of the VM tests except for those dealing with more complicated intercontract execution". However, we have decided not to use it for our checker due to three reasons: (*a*) the generated Coq code from Lem definitions is not "necessarily idiomatic" and thus it would generate a very complex `EVM` formalization in Coq that would make theorems harder to state and prove; (*b*) the author of the Lem definition states that "the Coq version of the `EVM` definition is highly experimental"; and (*c*) it is not kept up-to-date.

The other most developed implementation of the `EVM` semantics in Coq that we have found is [21]. It supports all the basic `EVM` bytecodes we consider in our checker, and looked promising as our departing point. The implementation uses *Bedrock Bit Vectors (bbv)* [7] for representing the `EVM` 256-bit values, as we use as well. It is not a full formalization of the `EVM` because it does not support calling or creation of smart contracts, but provides a function that simulates consequent application of opcodes to the given execution state, call info and Ethereum state mocks. The latter two pieces of information would add complexity and are not needed for our purpose. Therefore, we decided to develop our own `EVM` formalization in Coq (presented in Sec. 3) which builds upon some ideas of [21], but introduces only the minimal elements we need to handle the instructions supported by the checker. This way the proofs will be simpler and conciser.

3 EVM Semantics in Coq

Our EVM formalization is a concrete interpreter that executes a block of EVM instructions. For representing EVM words we use `EVMWord` that stands for the type "`word` 256" of the `bbv` library [7]. For representing instructions we use:

```
Inductive stack_op_instr :=        Inductive instr :=
  | ADD                              | PUSH (size: nat) (w: EVMWord)
  | MUL                              | POP
  | SUB                              | DUP (pos: nat)
  | DIV                              | SWAP (pos: nat)
  | NOT.                             | StackInstr (label: stack_op_instr).
```

Type `stack_oper_instr` defines instructions that operate only on the stack, i.e., each pops a fixed number of elements and pushes a single value back (see App. B for the full list). Type `instr` encapsulates this category together with the stack manipulation instruction (`PUSH`, etc). The type `block` stands for "`list instr`".

To keep the framework general, and simplify the proofs, the actual implementation of instructions from `stack_op_instr` are provided to the interpreter as input. For this, we use a map that associates instructions to implementations:

```
Inductive stack_operation :=
  | StackOp (comm: bool) (n : nat) (f : list EVMWord → option EVMWord).
Definition stack_op_map := map stack_oper_instr stack_operation.
```

The type `stack_operation` defines an implementation for a given operation: `comm` indicates if the operation is commutative; `n` is the number of stack elements to be removed and passed to the operation; and `f` is the actual implementation. The type `stack_op_map` maps keys of type `stack_oper_instr` to values of type `stack_operation`. Suppose `evm_add` and `evm_mul` are implementations of `ADD` and `MUL` (see App. C), the actual stack operations map is constructed as follows:

```
Definition evm_stack_opm : stack_op_map :=
  ADD |→i StackOp true 2 evm_add; MUL |→i StackOp true 2 evm_mul; ...
```

In addition, we require that the operations in the map to be valid with respect to the properties that they claim to satisfy (e.g., commutativity), and that when applied to the right number of arguments they should succeed (i.e., do not return `None`). We refer to this property as `valid_stack_op_map`.

An execution state (or simply state) includes only a stack (currently we support only stack operations) which is as a list of `EVMWord`, and the interpreter is a function that takes a block, an initial state, and a stack operations map, and iteratively executes each of the block's instructions:

```
Definition stack := list EVMWord.
Inductive state :=
  | ExState (stk: stack).
Fixpoint concr_int (p: block) (st: state) (ops: stack_op_map): option state := ...
```

The result can be either `Some st` or `None` in case of an error which are caused only due to stack overflow. In particular, we are currently not taking into account the amount of *gas* needed to execute the block. Note that our implementation follows the `EVM` semantics [36], and we plan to test it using EVM test suite.

4 Formal Verification of `EVM`-Optimizations in Coq

Two jump-free blocks `p1` and `p2` are equivalent wrt. to `k`, if for any initial stack of size `k`, the executions of `p1` and `p2` succeed and lead to the same state. Formally:

```
Definition sem_eq_blocks: (p1 p2: block) (k: nat) (ops: stack_op_map) : Prop :=
  ∀ (in_st: state) (in_stk: stack),
    get_stack in_st = in_stk → length in_stk = k →
        ∃ (out_st : state), concr_int p1 in_st ops = Some out_st ∧
                                  concr_int p2 in_st ops = Some out_st
```

Note that when `concr_int` returns `None` for both `p1` and `p2`, they are not considered equivalent because in the general case they can fail due to different reasons.

An `EVM` block equivalence checker is a function that takes two blocks, the size of the initial stack, and returns `true`/`false`. Its soundness is stated as follows:

```
Definition eq_block_chkr_snd (chkr : block → block → nat → bool) : Prop :=
  ∀ (p1 p2: block) (k: nat),
      chkr p1 p2 k = true → sem_equiv_blocks p1 p2 k evm_stack_opm
```

Given two blocks p_1 and p_2, checking their equivalence (in Coq) has the following components: (*i*) *Symbolic Execution (Sec. 4.1):* it is based on an interpreter that symbolically executes a block, wrt. an initial symbolic stack of size `k`, and generates a final symbolic stack. It is applied on both p_1 and p_2 to generate their corresponding symbolic output states $\mathtt{S}_1$ and $\mathtt{S}_2$. (*ii*) *Rule optimizations (Sec. 4.2):* it is based on simplification rules that are often applied by program optimizers, which rewrite symbolic states to equivalent "simpler" ones. This step simplifies $\mathtt{S}_1$ and $\mathtt{S}_2$ to $\mathtt{S}_1'$ and $\mathtt{S}_2'$. (*iii*) *Equivalence Checker (Sec. 4.3):* it receives the simplified symbolic states, and determines if they are equivalent for any concrete instantiation of the symbolic input stack. It takes into account, for example, the fact that some stack operations are commutative.

4.1 `EVM` Symbolic Execution in Coq

Symbolic execution takes an initial symbolic state (i.e., stack) $[s_0, \ldots, s_k]$, a block, and a map of stack operations, and generates a final symbolic state (i.e., stack) with symbolic expressions, e.g., $[5{+}s_0, s_1, s_2]$, representing the corresponding computations. In order to incorporate rule-based optimizations in a simple and efficient way, we want to avoid compound expressions such as $5 + (s_0 * s_1)$, and instead use temporal fresh variables together with a corresponding map that assigns them to simpler expressions. E.g, the stack $[5 + (s_0 * s_1), s_2]$ would be represented as a tuple $([e_1, s_2], \{e_1 \mapsto 5 + e_0, e_0 \mapsto s_0 * s_1\})$ where e_i are fresh variables. To achieve this, we define the *symbolic stack* as a list of elements that can be numeric constant values, initial stack variables or fresh variables:

```
Inductive sstack_val : Type :=
```

```
  | Val (val: EVMWord) | InStackVar (var: nat) | FreshVar (var: nat).
Definition sstack := list sstack_val.
```

and the map that assigns meaning to fresh variables is a list that maps each fresh variable to a `sstack_val`, or to a compound expression:

```
Inductive smap_val : Type :=
  | SymBasicVal (val: sstack_val)
  | SymOp (opcode : stack_op_instr) (args : list sstack_val).
Definition smap  := list (nat*smap_val).
```

Finally, a symbolic state is defined as a `SymState` term where `k` is the size of the initial stack, `maxid` is the maximum id used for fresh variables (kept for efficiency), `sstk` is a symbolic stack, and `m` is the map of fresh variables.

```
Inductive sstate : Type := | SymState (k maxid: nat) (sstk: sstack) (m: smap).
```

Example 1 (Symbolic execution). Given $p_1 \equiv$"`PUSH1 0x5 SWAP2 MUL ADD`" and $p_2 \equiv$ "`PUSH1 0x0 ADD MUL PUSH1 0x5 ADD`", symbolically executing them with `k`=3 we obtain the symbolic states represented by `sst1` $\equiv ([e'_1, s_2], \{e'_1 \mapsto e'_0 + 5, e'_0 \mapsto s_1 * s_0\})$ and `sst2` $\equiv ([e_2, s_2], \{e_2 \mapsto 5 + e_1, e_1 \mapsto e_0 * s_1, e_0 \mapsto 0 + s_0\})$.

Note that we impose some requirements on symbolic states to be valid. E.g., for any element $i \mapsto v$ of the fresh variables map, all fresh variables that appear in v have smaller indices than i. We refer to these requirements as `valid_sstate`.

Given a symbolic (final) state and a concrete initial state, we can convert the symbolic state into a concrete one by replacing each s_i by its corresponding value, and evaluating the corresponding expressions (following their definition in the stack operations map). We have a function to perform this evaluation that takes the stack operations map as input:

```
Definition eval_sstate (in_st: state) (sst: sstate) (ops : stack_op_map)
  : option state := ...
```

Our symbolic execution engine is a function that takes the size of the initial stack, a block, a map of stack operations, and generates a symbolic final state:

```
Definition sym_exec (p: block) (k: nat) (ops: stack_op_map) : option sstate := ...
```

Note that we do not pass an initial symbolic state, but rather we construct it inside using `k`. Also, the result can be `None` in case of failure (the causes are the same as those of `conc_interpreter`).

Soundness of `sym_exec` means that whenever it generates a symbolic state as a result, then the concrete execution from any stack of size `k` will succeed and produce a final state that agrees with the generated symbolic state:

```
Theorem sym_exec_snd:
  ∀ (p: block) (k: nat) (ops: stack_op_map) (sst: sstate),
     valid_stack_op_map ops →
     sym_exec p k ops = Some sst →
     valid_sstate sst ∧
     ∀ (in_st : state) (in_stk : stack),
       get_stack in_st = in_stk →
```

```
      length in_stk = k →
        ∃ (out_st : state),
          concr_int p in_st ops = Some out_st ∧
          eval_sstate in_st sst ops = Some out_st
```

4.2 Simplification Rules

To capture equivalence of programs that have been optimized according to "rule simplifications" (type (i) in Sec. 2) we need to include the same type of simplifications (see Fig. 1) in our framework. Without this, we will capture EVM-blocks equivalence only for "data-stack equivalence optimizations" (type (ii) in Sec. 2).

An *optimization function* takes as input a symbolic state, and tries to simplify it to an equivalent state. E.g, if a symbolic state includes $e_i \mapsto s_3 + 0$, we can replace it by $e_i \mapsto s_3$. The following is the type used for optimization functions:

```
Definition optim := sstate → sstate*bool.
```

Optimization functions never fail, i.e., in the worst case they return the same symbolic state. This is why the returned value includes a Boolean to indicate if any optimization has been applied, which is useful when composing optimizations later. The soundness of an optimization function can be stated as follows:

```
Definition optim_snd (opt: optim) : Prop :=
  forall (sst: sstate) (sst': sstate) (b: bool),
    valid_sstate sst → opt sst = (sst', b) →
      (valid_sstate sst' ∧
         forall (st st': state), eval_sstate st sst evm_stack_opm = Some st' →
                                 eval_sstate st sst' evm_stack_opm = Some st').
```

So far we have implemented and proven correct the most-used simplification rules of Fig. 1 (see App. A). E.g., there is an optimization function `optimize_add_zero` that rewrites expressions of the form $E + 0$ or $0 + E$ to E, and its soundness theorem is:

```
Theorem optimize_add_zero_snd: optim_snd optimize_add_zero.
```

Example 2. Consider again the blocks of Ex. 1. Using `optimize_add_zero` we can rewrite `sst2` to $\texttt{sst2}' \equiv ([e_2, s_2], \{e_2 \mapsto 5 + e_1, e_1 \mapsto e_0 * s_1, e_0 \mapsto s_0\})$, by replacing $e_0 \mapsto 0 + s_0$ by $e_0 \mapsto s_0$.

Note that the checker can be easily extended with new optimization functions, simply by providing a corresponding implementation and a soundness proof. Optimization functions can be combined to define *simplification strategies*, which are also functions of type `optim`. E.g., assuming that we have *basic* optimization functions $\texttt{f}_1$,...,$\texttt{f}_n$: (*1*) Apply $\texttt{f}_1$,...,$\texttt{f}_n$ iteratively such that in iteration i function $\texttt{f}_i$ is applied as many times as it can be applied. (*2*) Apply each $\texttt{f}_i$ once in some order and repeat the process as many times as it can be applied. (*3*) Use the simplifications that were used by the optimizer (it needs to pass these hints).

4.3 Stacks Equivalence Modulo Commutativity

We say that two symbolic stacks `sst1` and `sst2` are equivalent if for every possible initial concrete state `st` they evaluate to the same state. Formally:

```
Definition eq_sstate (sst1 sst2: sstate) (ops : stack_op_map) : Prop :=
  ∀ (st: state), eval_sstate st sst1 ops = eval_sstate st sst2 ops.
```

However, this notion of semantic equivalence is not computable in general, and thus we provide an effective procedure to determine such equivalence by checking that at every position of the stack both contain "similar" expressions:

```
Definition eq_sstate_chkr (sst1 sst2: sstate) (ops : stack_op_map) : bool := ...
```

To determine if two stack elements are similar, we follow their definition in the map if needed until we obtain a value that is not a fresh variable, and then either (1) both are equal constant values; (2) both are equal initial stack variables; or (3) both correspond to the same instruction and their arguments are (recursively) equivalent (taking into account the commutativity of operations).

Example 3. `eq_sstate_chkr` fails to prove equivalence of `sst1` and `sst2` of Ex. 1, because, when comparing e_2 and e'_1, it will eventually check if $0 + s_0$ and s_0 are equivalent. It fails because the comparison is rather "syntactic". However, it succeeds when comparing `sst1` and `sst2`' (Ex. 2), which is a simplification of `sst2`.

This procedure is an approximation of the semantic equivalence, and it can produce false negatives if two symbolic states are equivalent but are expressed with different syntactic constructions. However, it is sound:

```
Theorem eq_sstate_chkr_snd:
  ∀ (sst1 sst2: sstate) (ops : stack_op_map),
    valid_stack_op_map ops → valid_sstate sst1 → valid_sstate sst2 →
          eq_sstate_chkr sst1 sst2 ops = true → eq_sstate sst1 sst2 ops.
```

Note that we require the stack operations map to be valid in order to guarantee that the operations declared commutative in `ops` are indeed commutative. In order to reduce the number of false negatives, the simplification rules presented in Sec. 4.2 are very important to rewrite symbolic states into closer syntactic shapes that can be detected by `eq_sstate_chkr`.

Finally, given all the pieces developed above, we can now define the block equivalence checker as follows:

```
Definition evm_eq_block_chkr (opt: optim) (p1 p2: block) (k: nat) : bool :=
match sym_exec p1 k evm_stack_opm with
| None ⇒ false
| Some sst1 ⇒
    match sym_exec p2 k evm_stack_opm with
    | None ⇒ false
    | Some sst2 ⇒ let (sst2', _) := opt sst1 in
                  let (sst1', _) := opt sst2 in
                    eq_sstate_chkr sst1' sst2' evm_stack_opm
    end
end.
```

It symbolically executes `p1` and `p2`, simplifies the resulting symbolic states by applying optimization `opt`, and finally calls `eq_sstate_chkr` to check if the states are equivalent. The above checker is sound when `opt` is sound:

```
Theorem evm_eq_block_chkr_snd:
  ∀ (opt: optim), optim_snd opt → eq_block_chkr_snd (evm_eq_block_chkr opt)
```

5 Implementation and Experimental Evaluation

The different components of the tool have been implemented in Coq v8.15.2, together with complete proofs of all the theoretical results (more than 180 proofs in ∼7000 lines of Coq code). The code can be found at `https://github.com/costa-group/coq-evm/tree/v2023.01`. The tool currently includes 15 simplification rules from those of Fig. 1 (see App. A). We have tried our implementation on the outcome of two optimization tools: (1) the standalone `GASOL` optimizer and, (2) the optimizer integrated within the Solidity compiler `solc`. For (1), we have already fully automated the communication among the optimizer and checker and have been able to perform a thorough experimental evaluation. While in (2), the communication is more difficult to automate because the CFG of the original program can change after optimization, i.e., it can make cross-block optimization. Hence, in this case, we have needed human intervention to disable intra-block optimizations and obtain the blocks for the comparison (we plan to automate this usage in the future). For evaluating (2) we have used as benchmarks 1,280 blocks extracted from the smart contracts in the semantic test suite of the `solc` compiler [6], succeeding to prove equivalence on 1,045 out of them. We have checked that the fails are due to the use of optimization rules not yet implemented by us. Now we describe in detail the experimental evaluation on (1) for which we have used as benchmarks 147,798 blocks belonging to 96 smart contracts (see App. D).

`GASOL` allows enabling/disabling the application of simplification rules of Fig. 1, and choosing an optimization criteria: GAS consumption or bytes SIZE (of the code) [10]; combining these parameters we obtain 4 different sets of pairs-of-blocks to be verified by our tool. From these blocks, we consider only those that were actually optimized by `GASOL`, i.e., the optimized version is syntactically different from the original one. In all the cases, the average size of blocks is 8 instructions. Table 5.1 summarizes our results, where each row corresponds to one setting out of the 4 mentioned above: *Column 1* includes the optimization criteria; *Column 2* indicates if rule simplifications were applied by `GASOL`; *Column 3* indicates how many pairs-of-blocks were checked; *Columns 4-7* report the results of applying 2 versions of the checker, namely CHKR corresponds to the checker that only compares symbolic states and CHKR^s corresponds to the checker that also applies all the implemented rule optimizations iteratively as much as they can be applied (see Sect. 4.2). For each we report the number of instances it proved equivalent and the total runtime in seconds. The experiments have been performed on a machine with an Intel i7-4790 at 3.60 GHz and 16GB of RAM.

For sets in which `GASOL` does not apply simplification rules (marked with ×), both CHKR and CHKR^s succeed to prove equivalence of all blocks. When simplifications are applied (marked with ✓), CHKR^s succeeds in 99% of the blocks while CHKR ranges from 63% for GAS to 99% for SIZE. This difference is due to the fact that `GASOL` requires the application of rules to optimize more blocks

	SIMP	#blocks	CHKR Yes	CHKR Time	CHKR^s Yes	CHKR^s Time
GAS	×	36624	36624	2.60	36624	11.76
GAS	✓	43228	27149	4.69	43109	14.09
SIZE	×	35754	35754	2.57	35754	12.59
SIZE	✓	32192	31488	2.50	31798	12.17

Table 5.1. Summary of experiments using `GASOL`.

wrt. GAS ($\sim 37\%$ of the total) than wrt. SIZE ($\sim 1\%$). Moreover, all the blocks that CHKR^s cannot prove equivalent have been optimized by `GASOL` using rules which are not currently implemented in the checker, so we predict a success rate of 100% when all the rules in Fig. 1 are integrated. Regarding time, CHKR^s is 3–5 times slower than CHKR because of the overhead of applying rule optimizations, but it is still very efficient (all 147.798 instances are checked in 50.61 seconds). As a final comment, thanks to the checker we found a bug in the parsing component of `GASOL` that has been reported to its developers.

6 Conclusions, Related and Future Work

Our work provides the first tool able to formally verify the equivalence of jump-free `EVM` blocks and has required the development of all components within the verification framework. The implementation is not tied to any specific tool and could be easily integrated within any optimization tool. Ongoing work focuses on handling memory and storage optimizations. This extension needs to support the execution of memory/storage operations at the level of the concrete interpreter, and design an efficient data structure to represent symbolic memory/storage. Full handling of blockchain-specific opcodes is straightforward, it only requires adding the corrsponding implementations to the stack operations map `evm_stack_opm`. A more ambitious direction for future work is to handle cross-block optimizations.

There are two approaches to verify program optimizations, (1) verify the correctness of the optimizations and develop a *verified tool*, e.g., this is the case of optimizations within the CompCert certified compiler [22] and a good number of optimizations that have been formally verified in Coq [25,17,30,31,12], (2) or use a *translation validation* approach [32,33,34,18] in which rather than verifying the tool, each of the compiled/optimized programs are formally checked to be correct using a verified checker. We argue that translation validation [32] is the most appropriate approach for verifying `EVM` optimizations because: (i) `EVM` compilers (together with their built-in optimizers) are continuously evolving to adjust to modifications in the rather new blockchain programming languages, (ii) existing `EVM` optimizers use external components such as SMT solvers to search for the optimized code and verifying an SMT solver would require a daunting effort, (iii) we aim at generality of our tool rather than restricting ourselves to a specific optimizer and, as already explained, the design of our checker has been done having generality and extensibility in mind, so that new optimizations can be easily incorporated. Finally, it is worth mentioning the KEVM framework [19], which in principle could be the basis for verifying optimizations as well. However, we have chosen to develop it in Coq due to its maturity.

References

1. https://www.certora.com/.
2. https://veridise.com/.
3. https://www.apriorit.com/.
4. https://consensys.net/.
5. https://www.dedaub.com/.
6. https://github.com/ethereum/solidity/tree/develop/test/libsolidity/semanticTests/externalContracts.
7. Bedrock Bit Vectors (bbv), 2018. https://github.com/mit-plv/bbv.
8. PausableERC20 Contract, 2020. https://etherscan.io/address/0x32E6C34Cd57087aBBD59B5A4AECC4cB495924356.
9. The solc optimizer, 2021. https://docs.soliditylang.org/en/v0.8.7/internals/optimizer.html.
10. Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, and Albert Rubio. A max-smt superoptimizer for EVM handling memory and storage. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 201–219. Springer, 2022.
11. Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria A. Schett. Synthesis of super-optimized smart contracts using max-smt. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 177–200. Springer, 2020.
12. Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. Formally verified speculation and deoptimization in a JIT compiler. *Proc. ACM Program. Lang.*, 5(POPL):1–26, 2021.
13. Thomas Bernardi, Nurit Dor, Anastasia Fedotov, Shelly Grossman, Alexander Nutz, Lior Oppenheim, Or Pistiner, Mooly Sagiv, John Toman, and James Wilcox. Preventing reentrancy bugs - another use case for formal verification, 2020. https://www.certora.com/blog/reentrancy.html.
14. Alina Bizga. A hackers' dream payday: Ledf.me and uniswap lose $25 million worth of cryptocurrency. https://securityboulevard.com/2020/04/a-hackers-dream-payday-ledf-me-and-uniswap-lose-25-million-worth-of-cryptocurrency/, 2020. [Online; accessed 11-May-2020].
15. Vitalik Buterin. CRITICAL UPDATE Re: DAO vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/, 2016. [Online; accessed 2-July-2017].
16. Phil Daian. Analysis of the DAO exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 2016.
17. Delphine Demange, David Pichardie, and Léo Stefanesco. Verifying fast and sparse ssa-based optimizations in coq. In Björn Franke, editor, *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9031 of *Lecture Notes in Computer Science*, pages 233–252. Springer, 2015.
18. Léo Gourdin and Sylvain Boulmé. Certifying assembly optimizations in Coq by symbolic execution with hash-consing. page 2, 2021.

19. Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 204–217. IEEE Computer Society, 2018.
20. Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pages 520–535, Cham, 2017. Springer International Publishing.
21. ivan71kmayshan27. Coq formalisation of the ethereum virtual machine (wip). https://github.com/ivan71kmayshan27/coq-evm, 2020. Accessed: 2022-06-23.
22. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
23. Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Practical Verification of Peephole Optimizations with Alive. *Commun. ACM*, 61(2):84–91, January 2018.
24. Henry Massalin. Superoptimizer - A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–126, 1987.
25. David Monniaux and Cyril Six. Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion. In Jörg Henkel and Xu Liu, editors, *LCTES '21: 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, Virtual Event, Canada, 22 June, 2021*, pages 85–96. ACM, 2021.
26. Dominic P Mulligan, Scott Owens, Kathryn E Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. *ACM SIGPLAN Notices*, 49(9):175–188, 2014.
27. Julian Nagele and Maria A Schett. Blockchain superoptimizer. In *Preproceedings of 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019)*, 2019. https://arxiv.org/abs/2005.05912.
28. Daniel Palmer. Spankchain loses $40k in hack due to smart contract bug. https://www.coindesk.com/spankchain-loses-40k-in-hack-due-to-smart-contract-bug, 2018. [Online; accessed 11-May-2020].
29. Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. Souper: A Synthesizing Superoptimizer. *arXiv:1711.04422 [cs]*, November 2017.
30. Cyril Six, Sylvain Boulmé, and David Monniaux. Certified and efficient instruction scheduling: application to interlocked VLIW processors. *Proc. ACM Program. Lang.*, 4(OOPSLA):129:1–129:29, 2020.
31. Cyril Six, Léo Gourdin, Sylvain Boulmé, David Monniaux, Justus Fasse, and Nicolas Nardino. Formally verified superblock scheduling. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 40–54. ACM, 2022.
32. Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 17–27. ACM, 2008.

33. Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 316–326. ACM, 2009.
34. Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 83–92. ACM, 2010.
35. Cooper Turley. imBTC uniswap pool drained for \$300k in ETH. https://defirate.com/imbtc-uniswap-hack/, 2020. [Online; accessed 11-May-2020].
36. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger (Berlin version 8fea825 – 2022-08-22), 2022.

A Rule simplifications

Fig. 1 includes rule simplifications that are applied by tools like `GASOL` and the Solidity compiler `solc`. Those rules integrated in our checker are marked with *.

(1*) $\mathbf{OP(X_{int}, Y_{int}) = eval(OP, X_{int}, Y_{int})}$
(2*) $\mathbf{OP(X_{int}) = eval(OP, X_{int})}$
(3*) $\mathbf{ADD(X, 0) = X}$
(4) $SUB(X, 0) = X$
(5*) $\mathbf{SUB(X, X) = 0}$
(6*) $\mathbf{MUL(X, 0) = 0}$
(7*) $\mathbf{MUL(X, 1) = X}$
(8) $MUL(SHL(X, 1), Y) = SHL(X, Y)$
(9) $MUL(X, SHL(Y, 1)) = SHL(Y, X)$
(10) $DIV(X, X) = 1$
(11*) $\mathbf{DIV(X, 1) = X}$
(12) $DIV(X, 0) = 0$
(13) $DIV(X, SHL(Y, 1)) = SHR(Y, X)$
(14) $MOD(X, 1) = 0$
(15) $MOD(X, X) = 0$
(16) $MOD(X, 0) = 0$
(17) $EXP(X, 0) = 1$
(18) $EXP(X, 1) = X$
(19) $EXP(1, X) = 1$
(20) $EXP(0, X) = ISZERO(X)$
(21) $EXP(2, X) = SHL(X, 1)$
(22) $GT(0, X) = 0$
(23*) $\mathbf{GT(1, X) = ISZERO(X)}$
(24) $GT(X, X) = 0$
(25) $LT(X, 0) = 0$
(26*) $\mathbf{LT(X, 1) = ISZERO(X)}$
(27) $LT(X, X) = 0$
(28) $EQ(X, X) = 1$
(29*) $\mathbf{EQ(X, 0) = ISZERO(X)}$
(30) $EQ(1, ISZERO(X)) = ISZERO(X)$
(31) $ISZERO(SUB(X, Y)) = EQ(X, Y)$
(32) $ISZERO(GT(X, 0)) = ISZERO(X)$
(33) $ISZERO(LT(0, X)) = ISZERO(X)$
(34*) $\mathbf{ISZERO(ISZERO(ISZERO(X))) = ISZERO(X)}$
(35) $ISZERO(XOR(X, Y)) = EQ(X, Y)$
(36) $ISZERO(ISZERO(GT(X, Y))) = GT(X, Y)$
(37) $ISZERO(ISZERO(LT(X, Y))) = LT(X, Y)$
(38) $ISZERO(ISZERO(EQ(X, Y))) = EQ(X, Y)$
(39) $SHL(X, 0) = 0$
(40) $SHL(0, X) = X$
(41) $SHR(0, X) = X$
(42) $SHR(X, 0) = 0$
(43*) $\mathbf{NOT(NOT(X)) = X}$
(44) $XOR(X, X) = 0$
(45) $XOR(X, 0) = X$
(46) $XOR(X, XOR(X, Y)) = Y$
(47*) $\mathbf{OR(X, 0) = X}$
(48) $OR(2^{256} - 1, X) = 2^{256} - 1$
(49) $OR(X, X) = X$
(50) $OR(X, AND(X, Y)) = X$
(51) $OR(OR(X, Y), Y) = OR(X, Y)$
(52) $OR(OR(Y, X), Y) = OR(Y, X)$
(53) $OR(X, NOT(X)) = 2^{256} - 1$
(54) $AND(X, 0) = 0$
(55) $AND(X, X) = X$
(56) $AND(2^{256} - 1, X) = X$
(57*) $\mathbf{AND(AND(X, Y), Y) = AND(X, Y)}$
(58*) $\mathbf{AND(AND(Y, X), Y) = AND(Y, X)}$
(59) $AND(X, OR(X, Y)) = X$
(60) $AND(X, NOT(X)) = 0$
(61) $AND(ORIGIN, 2^{160} - 1) = ORIGIN$
(62) $AND(CALLER, 2^{160} - 1) = CALLER$
(63) $AND(ADDRESS, 2^{160} - 1) = ADDRESS$
(64) $AND(COINBASE, 2^{160} - 1) = COINBASE$
(65) $BALANCE(ADDRESS) = SELFBALANCE$

Fig. 1. Simplification rules

B Supported Instructions

In Sec. 3 we have discussed a limited number of instructions that are included in the type `stack_op_instr`. Next we overview the full list of supported instructions.

The following stack operations are supported in the checker:

ADD	MUL	NOT	SUB	DIV	SDIV	MOD
SMOD	ADDMOD	MULMOD	EXP	SIGNEXTEND	LT	GT
SLT	SGT	EQ	ISZERO	AND	OR	XOR
BYTE	SHL	SHR	SAR			

They all perform computations that depend only on values stored on the stack, and the result is pushed again to the stack. In addition, in order to increase the set of benchmarks that we can handle, the following are also supported, even though they depend on values stored in the memory/store/contract/etc, in a sound way:

SHA3	KECCAK256	ADDRESS	BALANCE	ORIGIN
CALLER	CALLVALUE	CALLDATALOAD	CALLDATASIZE	CODESIZE
GASPRICE	EXTCODESIZE	RETURNDATASIZE	EXTCODEHASH	BLOCKHASH
COINBASE	TIMESTAMP	NUMBER	DIFFICULTY	GASLIMIT
CHAINID	SELFBALANCE	BASEFEE	SLOAD	MLOAD
PC	MSIZE	GAS		

The support is done by corresponding implementations that always return 0. This is sound for now since (1) they are not used in rule simplifications, and thus they are never executed; and (2) we do not have any instruction that modifies the memory/store. Recall that when comparing symbolic states, if two calls to the same instruction are compared then we require their arguments to be equal and we do not inspect the returned value. For now, in the implementations, they are defined as `stack_op_instr` but this will change once other instruction categories are formalized.

C Missing code

```
Definition evm_add (args: list EVMWord) : option EVMWord :=
match args with
| [a; b] ⇒ Some (wplus a b) (* wplus is part of the bbv library *)
| _ ⇒ None
end.
```

```
Definition stack_op_map_comm (ops: stack_op_map) : Prop :=
  ∀ (instr : stack_oper_instr) (f: list EVMWord → option EVMWord),
    ops instr = Some (StackOp true 2 f) →
      ∀ (a b: EVMWord), f [a;b] = f [b;a].

Definition coherent_stack_op_map (ops: stack_op_map) : Prop :=
```

```
∀ (instr: stack_op_instr) (comm: bool) (n: nat)
  (f: list EVMWord → option EVMWord),
    ops instr = Some (StackOp comm n f) →
      ∀ (l: list EVMWord),
        length l = n → ∃ (v: EVMWord), f l = Some v.

Definition valid_stack_op_map (ops: stack_op_map) : Prop :=
  stack_op_map_comm ops ∧ coherent_stack_op_map ops
```

D Smart contracts used in the experiments

For the experimental evaluation we have extracted all the blocks in the following 96 smart contracts from the Ethereum blockchain. We identify the smart contracts by their addresses.

1. 0x0621213b273bff05d679d9b1c68ec18cf989168f
2. 0x0edc5ac3da3df2e4643aca1a777ca9eb6656117a
3. 0x0f066b014adb057cdfc6c587965fbaa14151dfa5
4. 0x100739d55a4e8361dcca7e872426c4b6aadeb201
5. 0x16c19aaae850bb0282b38686fb071fe37edecf1f
6. 0x16d1884381d94b372e6020a28bf41bbabe8c1f26
7. 0x1c52b09ccddd1b6999400b038c7e0680eaf03dcd
8. 0x1Ee8923Db533Ecb7A4650cCc8829D6F114D718f9
9. 0x269028c988db0e6786de1f4ff66af7c033d0f6c8
10. 0x2ccc239e97d01ad4c39a323327bc1a1f4cb43076
11. 0x2e12AE85aF4121156F62ad4D059415C746fe615c
12. 0x34662bf3ad9b3a70ea5b46ad81f4e9cab4d89a7f
13. 0x359651fb6636cb650Fa47F11C9D35533BbFF8158
14. 0x3948b7b6b8812439ddcbc8fa42cac8e213191792
15. 0x3ae30bb991be0d54fddfedc7d6556e20daa97c71
16. 0x3e456b66425f02bfe3896c1cc3b8513ff660b4bf
17. 0x4152e8133d7927988101378910024 6a907884b9e
18. 0x4226cac9567e991f956f644b656ce4aa0599c63e
19. 0x44f8217a9dbb45ef2491da6aa18826bd6ded7847
20. 0x450f184242894d068a71d3abfa296a73df1e192c
21. 0x4757388aa7e3490106092bde16c23e2858c7d405
22. 0x47B51F81E03fB068d776CcB78b08F59e5256B944
23. 0x49173F2BF921Ce4124A8C6aBad3c5875Ff40D951
24. 0x49566ab7ef0d4da06a3117e9e4fb3e9947abaf96
25. 0x4d2d88d73ab4062d61b1eb68b5808b9176cef271
26. 0x4d37D0aB328e1D449Ea8CFc3b0B7364B398c41E0
27. 0x4E4bd1f64232450bEa37c4CB76D4b4cda3d46DAa
28. 0x4f73c17195d0f77c1fc4175345b9251a9fb21849
29. 0x4f89a0d9d868a39ec7024828dcaaae140a1a7ff3
30. 0x5036f390F631f66284253864aE351B0297E32f03
31. 0x50b6c438f108b5c0145142f54d538e704c55995b

32. 0x54adf7604d25ac9476fc467e93dfdb29af1077ee
33. 0x58760b75093a8462eb2eab2c5769ba5c0b18fa68
34. 0x5d2fdd14e44b090f2eef03c715d414039f86d7bd
35. 0x60e600a4d09297f9e9bb6eb90373f48e7830e69c
36. 0x6365303A5E1C1327b36bDa8C22440be94eCCbcA1
37. 0x64b88f10faf1603b70fb7370a00c43369f329515
38. 0x697720ee431148a586a546551de6c4d575e4d8d0
39. 0x6cd5a65e85c9603df74d4311d76145820556548a
40. 0x6e53a6441b24cb773adcc6e6f9d43e956e7c9a6e
41. 0x70001ba1ba4d85739e7b6a7c646b8aba5ed6c888
42. 0x702197775Ab2B462Af51Ba11b38d103AaA0bb443
43. 0x72BD2930663b30dBA3cd362bc1f8C2251E24C73A
44. 0x766a339751Df1705364D961b4f7f87309F210355
45. 0x7a741d7ff3da76d722fa4a37455f099efd0ef168
46. 0x7bd251d43d8ee259acde7788ec93b7f3d6626dd2
47. 0x7dDA9F944c3Daf27fbe3B8f27EC5f14FE3fa94BF
48. 0x7F197F94cA6e57Fd983cE0fa29710cfA3b842bf8
49. 0x89872650fa1a391f58b4e144222bb02e44db7e3b
50. 0x8EfbD976709c2bD46bdaf0c3Db83E875F1451BAE
51. 0x8f093895cd4c54eab897c6377e1bf85fe5b4e948
52. 0x8f3b62dd6a9bf905516f433c214753934b337e05
53. 0x90f24a2432a8b2e87b5a2026855181317890d129
54. 0x949205a8e58bd1e5eb043c6379d1e7564a85039a
55. 0x94a79038D97e22AC47C9Aa41624f948BDd7ac27D
56. 0x99E2C293A8A6c3dAE6A591CEA322D0c3Cd231B2C
57. 0xa403f555e419e56F49ba90022f7E7d0d3e86522D
58. 0xa7b30042c7e798d0be8e466bf879388acddc526f
59. 0xaa30979b30523bff7ca2ba434d126d15ad5b0905
60. 0xAa7B19b68a1da16f272564e74b0e99f969c4DF6a
61. 0xAbF18841Dca279a030bd9A9122F4460Da57ad547
62. 0xAbf52Fc6e5C0e6E44Daac7C6ca79498302D9B0CA
63. 0xb4e2ebaf639fd03aebe85bd0960b49ade9879b0f
64. 0xb4feb1f99fc9e2688729fc899e1ee3631bbebded
65. 0xb5615b9799427280cbc34a33233ef59b6409a711
66. 0xb595e208833164d43a08ce529acc2b803d33c30e
67. 0xb6105c0fa743290f94da9bf304ac45c19f4b2851
68. 0xb8ec5b27de7d971d74e8531baa27853cffdfae1d
69. 0xbb4ed94d1f743a8bba6318c821a7e9cf1d632c96
70. 0xbdafddc47d1cbac27f80a918908922aa6bf4b5bc
71. 0xbdf9d5752ec89a3b7c3b7ffe31f5bd565016c221
72. 0xc581bced4dedab50e8bfdcc67b2a36b92e013d78
73. 0xcbdbc7c264c1abff6646bdd5ba13f1664823b0cd
74. 0xcc07e50a953c8c61f5dc077ed171e46210e9783e
75. 0xcCFFC73347B05cBDCCe7c3de2a0AdEcDAa8AEf51
76. 0xcD097cdB473286a10Cf19CbA9597E4400e8B6943
77. 0xcfc131c7810f9f7ec859bd3dd020bdb4bb06a5a8

78. 0xd2947e1e2ea5c4cd14aaa2b7492549129b087daa
79. 0xda44b167ca409b7fc51ccbd6ef3338b8e19999a8
80. 0xde1972989f633f10b6e6dc581b785c4618aa8490
81. 0xdfd5235a6d3e184ba27307d7d21ae9b425ff4e6d
82. 0xe45D283123607B7D97856d49C965faa40542BA94
83. 0xe7a2241e92c7b7299452e63d53af6692dfcd0367
84. 0xe8d2f4b9edbb0244167339c3a8daa6d82d959e72
85. 0xEadC2a6fff036C12e62A74392d4c6CA77A5Ea007
86. 0xeb453a070c20e79ff148e0506bd02c30b577af43
87. 0xEf78B55bD7bC090F809535f3B32Bcf1E050815df
88. 0xF0B0ccED14b2d1D47C351F5Bc0B33AA79470997e
89. 0xf1cb8f9738adff8c280d6eae8e2285a839b79f80
90. 0xF2281cA8693B1d35D7a73700909ec8535A57156D
91. 0xf508bda527d4ef9db712eb0100f1cd8f573bbe88
92. 0xf66ff968773e45dad3e1ac13ffbb63fae0eb1624
93. 0xf7a84edAc5539b75AFaaA04f1103dBf9Db4b09c6
94. 0xfa1c9bf3de714059b3c019facdcaef785cab098e
95. 0xfc21969625ae8933e85b49df3cc28aa7092fcfc7
96. 0xfeff9661617cbba5a2ed3a69000f4bf1e266be71