Lecture Notes in Computer Science **13966**

Constantin Enea · Akash Lal
Editors

Computer Aided Verification

35th International Conference, CAV 2023
Paris, France, July 17–22, 2023
Proceedings, Part III

Springer

Editors
Constantin Enea
LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris
Palaiseau, France

Akash Lal
Microsoft Research
Bangalore, India

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-37708-2 ISBN 978-3-031-37709-9 (eBook)
https://doi.org/10.1007/978-3-031-37709-9

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Formally Verified EVM Block-Optimizations

Elvira Albert[1], Samir Genaim[1], Daniel Kirchner[2,3], and Enrique Martin-Martin[1(✉)]

[1] Complutense University of Madrid, Madrid, Spain
{elvira,samir.genaim}@fdi.ucm.es, emartinm@ucm.es
[2] Ethereum Foundation, Zug, Switzerland
daniel.kirchner@ethereum.org
[3] University of Bamberg, Bamberg, Germany

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.

Artifact: https://doi.org/10.5281/zenodo.7863483

Keywords: Coq · Ethereum Virtual Machine · Smart Contracts · Optimization · Theorem Proving

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., [14–17,30,37]). There is hence huge interest and investment in guaranteeing their correctness, e.g., Certora [1], Veridise [2], aprioriort [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

This work was funded partially by the Ethereum Foundation under Grant ID FY22-0698 and the Spanish MCI, AEI and FEDER (EU) project PID2021-122830OB-C41.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13966, pp. 176–189, 2023.
https://doi.org/10.1007/978-3-031-37709-9_9

emerged in the last few years (e.g., `ebso` [29], `SYRUP` [12], `GASOL` [11], 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 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 [11,12,29] 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 [11], 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 `SWAP`s 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 $\mathtt{S}_\mathtt{I}$ and $\mathtt{S}_\mathtt{O}$, (2) a series of simplification rules, which transform $\mathtt{S}_\mathtt{I}$ and $\mathtt{S}_\mathtt{O}$ into

equivalent ones $\mathtt{S}'_\mathtt{I}$ and $\mathtt{S}'_\mathtt{0}$, (3) a checker of symbolic states in Coq to decide if two symbolic states $\mathtt{S}'_\mathtt{I}$ and $\mathtt{S}'_\mathtt{0}$ are semantically equivalent.

2 Background

The Ethereum VM (EVM) [38] 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 Sect. 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* [26] 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 [25,31] 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 like `ADD(X,0)=X` or `NOT(NOT(X))=X` (see a complete list of these rules in App. A in [10]); and *(ii)* Stack-data optimizations which consist in searching for alternative stack operations that lead to an output stack with exactly the same data.

Example 1 (Intra-block optimizations). The rule-based optimization (i) `X+0` $\rightarrow$ `X` simplifies the block "`PUSH1 0x5, PUSH1 0x0, ADD`" to "`PUSH1 0x5`". On the other hand, stack-data optimizations (ii) can optimize to "`ADD, DUP1`" the block "`DUP2, DUP2, ADD, SWAP2, ADD`", as duplicating the operands and repeating the `ADD` operation is the same as duplicating the result. Unlike rule-based optimization, stack-data optimizations cannot be expressed as simple patterns that can be easily recognized.

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` [29] 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` [12] and `GASOL` [11] 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 (Sect. 4.2) while (ii) are implicit within the symbolic execution (Sect. 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 [22], which is a definition of the EVM semantics in the Lem [28] language that can be exported to interactive theorem provers like Isabelle/HOL or Coq. According to the comparison in [21], 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 [23]. 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 Sect. 3) which builds upon some ideas of [23], 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 :=
  | ADD
  | MUL
  | SUB
  | DIV
  | NOT.
```

```
Inductive instr :=
  | PUSH (size: nat) (w: EVMWord)
  | POP
  | DUP (pos: nat)
  | SWAP (pos: nat)
  | 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 in [10] 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 in [10]), the actual stack operations map is constructed as:

```
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 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. Our implementation follows the `EVM` semantics [38], considering the simplicity of the supported operations, the concrete interpreter is a minimal trusted computing base. In the future, we plan to test it using the EVM test suite.

4 Formal Verification of `EVM`-Optimizations in Coq

Two jump-free blocks `p1` and `p2` are equivalent *wrt.* to an initial stack size `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. Note also that EVM operations are deterministic, so if `concr_int` evaluates to a sucessful final state `out_st` it will be unique.

An `EVM` block equivalence checker is a function that takes two blocks, the size of the initial stack, and returns `true`/`false`. Providing the size `k` of the initial

stack is not a limitation of the checker, as this information is statically known in advance. Note that the maximum stack size in EVM is bounded by 1024, and that if the execution (of one or both blocks) *wrt.* to this concrete initial stack size leads to under/over stack overflow they cannot be reported equivalent. The soundness of the equivalence checker 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 (Sect. 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 (Sect. 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 (Sect. 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 2 (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 Sect. 2) we need to include the same type of simplifications (see App. A in [10]) in our framework. Without this, we will capture `EVM`-blocks equivalence only for "data-stack equivalence optimizations" (type (ii) in Sect. 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').
```

We have implemented and proven correct the most-used simplification rules (see App. A in [10]). 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 3. Consider again the blocks of Example 2. 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). E.g., the stack elements (viewed as terms) DIV(MUL(s_0,ADD(s_1,s_2)),0x16) and DIV(MUL(ADD(s_2,$s1$),s_0),0x16) are considered equivalent because the operations ADD and MUL are commutative.

Example 4. `eq_sstate_chkr` fails to prove equivalence of `sst1` and `sst2` of Example 2, 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'` (Example 3), 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 Sect. 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. Note that it is important to apply the optimization rules to both blocks, as the checker might apply optimization rules that were not applied by the external optimizer. This would lead to equivalent symbolic states with different shapes that will not be detected by the symbolic state equivalence checker.

Table 1. Summary of experiments using `GASOL`.

		#blocks	CHKR		CHKR^s	
	SIMP		Yes	Time	Yes	Time
GAS	×	36624	36624	2.60	36624	11.76
	✓	43228	27149	4.69	43109	14.09

		#blocks	CHKR		CHKR^s	
	SIMP		Yes	Time	Yes	Time
SIZE	×	35754	35754	2.57	35754	12.59
	✓	32192	31488	2.50	31798	12.17

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 source code, executables and benchmarks can be found at https://github.com/costa-group/forves/tree/stack-only and the artifact at https://doi.org/10.5281/zenodo.7863483. The tool currently includes 15 simplification rules (see App. A in [10]). We have tried our implementation on the outcome of two optimization tools: (1) the standalone `GASOL` optimizer and, (2) the optimizer integrated within the official 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. As these blocks are obtained from the test suite of the official `solc` Solidity compiler, optimized using the `solc` optimizer, the good results on this set suggest the validity can be generalized to other optimizers. 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 in [10]).

`GASOL` allows enabling/disabling the application of simplification rules and choosing an optimization criteria: GAS consumption or bytes SIZE (of the code) [11]; 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 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 *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 App. A in [10] 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 treated the SGT bytecode as GT. The bug was directly reported to the GASOL developers and is already fixed [19].

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 corresponding 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 [24] and a good number of optimizations that have been formally verified in Coq [13,18,27,32,33], (2) or use a *translation validation* approach [20,34–36] 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 [34] 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 [21], 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. Albert, E., Genaim, S., Kirchner, D., Martin-Martin, E.: Formally Verified EVM Block-Optimizations (Extended Version). https://costa.fdi.ucm.es/papers/costa/AlbertGKMM23_extended.pdf
11. Albert, E., Gordillo, P., Hernández-Cerezo, A., Rubio, A.: A Max-SMT superoptimizer for EVM handling memory and storage. In: TACAS 2022. LNCS, vol. 13243, pp. 201–219. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_11
12. Albert, E., Gordillo, P., Rubio, A., Schett, M.A.: Synthesis of super-optimized smart contracts using max-SMT. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 177–200. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_10
13. Barrière, A., Blazy, S., Flückiger, O., Pichardie, D., Vitek, J.: Formally verified speculation and deoptimization in a JIT compiler. Proc. ACM Program. Lang. **5**(POPL), 1–26 (2021). https://doi.org/10.1145/3434327
14. Bernardi, T., et al.: Preventing reentrancy bugs - another use case for formal verification (2020). https://www.certora.com/blog/reentrancy.html
15. Bizga, A.: A hackers' dream payday: Ledf.me and uniswap lose $25 million worth of cryptocurrency (2020). https://securityboulevard.com/2020/04/a-hackers-dream-payday-ledf-me-and-uniswap-lose-25-million-worth-of-cryptocurrency/. [Online; accessed 11-May-2020]
16. Buterin, V.: CRITICAL UPDATE Re: DAO vulnerability (2016). https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/. Accessed 2-July-2017

17. Daian, P.: Analysis of the DAO exploit (2016). http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
18. Demange, D., Pichardie, D., Stefanesco, L.: Verifying fast and sparse SSA-based optimizations in Coq. In: Franke, B. (ed.) CC 2015. LNCS, vol. 9031, pp. 233–252. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46663-6_12
19. elexcere: SGT and GT order when parsing. https://github.com/costa-group/gasol-optimizer/commit/fd78e126c23f192ed6c54aea713b5c94d3c943f5
20. Gourdin, L., Boulmé, S.: Certifying assembly optimizations in Coq by symbolic execution with hash-consing, p. 2 (2021)
21. Hildenbrandt, E., et al.: 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, pp. 204–217. IEEE Computer Society (2018). https://doi.org/10.1109/CSF.2018.00022
22. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
23. ivan71kmayshan27: Coq formalisation of the Ethereum Virtual Machine (WIP) (2020). https://github.com/ivan71kmayshan27/coq-evm
24. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
25. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Practical verification of peephole optimizations with alive. Commun. ACM **61**(2), 84–91 (2018). https://doi.org/10.1145/3166064
26. Massalin, H.: 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), pp. 122–126 (1987). https://dl.acm.org/citation.cfm?id=36194
27. Monniaux, D., Six, C.: Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion. In: Henkel, J., Liu, X. (eds.) LCTES '21: 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, Virtual Event, Canada, 22 June, 2021, pp. 85–96. ACM (2021). https://doi.org/10.1145/3461648.3463850
28. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: reusable engineering of real-world semantics. ACM SIGPLAN Notices **49**(9), 175–188 (2014)
29. Nagele, J., Schett, M.A.: Blockchain superoptimizer. In: Preproceedings of 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019) (2019). https://arxiv.org/abs/2005.05912
30. Palmer, D.: Spankchain loses $40k in hack due to smart contract bug (2018). https://www.coindesk.com/spankchain-loses-40k-in-hack-due-to-smart-contract-bug. Accessed 11 May 2020
31. Sasnauskas, R., et al.: Souper: A Synthesizing Superoptimizer. arXiv:1711.04422 [cs], November 2017
32. Six, C., Boulmé, S., Monniaux, D.: Certified and efficient instruction scheduling: application to interlocked VLIW processors. Proc. ACM Program. Lang. **4**(OOPSLA), 129:1–129:29 (2020). https://doi.org/10.1145/3428197

33. Six, C., Gourdin, L., Boulmé, S., Monniaux, D., Fasse, J., Nardino, N.: Formally verified superblock scheduling. In: Popescu, A., Zdancewic, S. (eds.) CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17–18, 2022, pp. 40–54. ACM (2022). https://doi.org/10.1145/3497775.3503679
34. Tristan, J., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7–12, 2008, pp. 17–27. ACM (2008). https://doi.org/10.1145/1328438.1328444
35. Tristan, J., Leroy, X.: Verified validation of lazy code motion. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15–21, 2009, pp. 316–326. ACM (2009). https://doi.org/10.1145/1542476.1542512
36. Tristan, J., Leroy, X.: A simple, verified validator for software pipelining. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17–23, 2010, pp. 83–92. ACM (2010). https://doi.org/10.1145/1706299.1706311
37. Turley, C.: imBTC uniswap pool drained for $300k in ETH (2020). https://defirate.com/imbtc-uniswap-hack/. Accessed 11 May 2020
38. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (Berlin version 8fea825 - 2022–08-22) (2022)