

Blockchain Superoptimizer^{*}

Julian Nagele¹[0000–0002–4727–4637] and Maria A Schett²[0000–0003–2919–5983]

¹ Queen Mary University of London, UK mail@jnagele.net

² University College London, UK mail@maria-a-schett.net

Abstract. In the blockchain-based, distributed computing platform Ethereum, programs called smart contracts are compiled to bytecode and executed on the Ethereum Virtual Machine (EVM). Executing EVM bytecode is subject to monetary fees—a clear optimization target. Our aim is to superoptimize EVM bytecode by encoding the operational semantics of EVM instructions as SMT formulas and leveraging a constraint solver to automatically find cheaper bytecode. We implement this approach in our EVM Bytecode SuperOptimizer `ebso` and perform two large scale evaluations on real-world data sets.

Keywords: Superoptimization, Ethereum, Smart Contracts, SMT

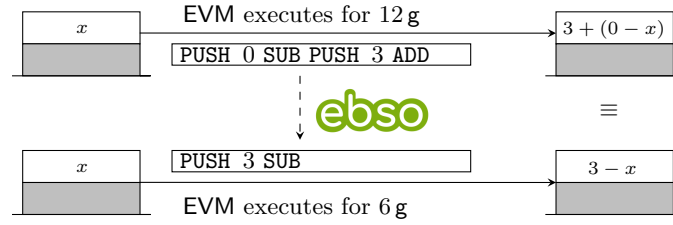
1 Introduction

Ethereum is a blockchain-based, distributed computing platform featuring a quasi-Turing complete programming language. In **Ethereum**, programs are called smart contracts, compiled to bytecode and executed on the Ethereum Virtual Machine (EVM). In order to avoid network spam and to ensure termination, execution is subject to monetary fees. These fees are specified in units of *gas*, *i.e.*, any instruction executed on the EVM has a cost in terms of gas, possibly depending on its input and the execution state.

Example 1. Consider the expression $3 + (0 - x)$, which corresponds to the program `PUSH 0 SUB PUSH 3 ADD`. The EVM is a stack-based machine, so this program takes an argument x from the stack to compute the expression above. However, clearly one can save the `ADD` instruction and instead compute $3 - x$, *i.e.*, optimize the program to `PUSH 3 SUB`. The first program costs 12 g to execute on the EVM, while the second costs only 6 g.

We build a tool that automatically finds this optimization and similar others that are missed by state-of-the-art smart contract compilers: the **EVM** bytecode superoptimizer `ebso`. The use of `ebso` for Example 1 is sketched in Figure 1. To find these optimizations, `ebso` implements *superoptimization*. Superoptimization is often considered too slow to use during software development except for special circumstances. We argue that compiling smart contracts is such a circumstance.

^{*} This research is supported by the UK Research Institute in Verified Trustworthy Software Systems and partially supported by funding from Google.

Fig. 1: Overview over `ebso`.

Since bytecode, once it has been deployed to the blockchain, cannot change again, spending extra time optimizing a program that may be called many times, might well be worth it. Especially, since it is very clear what “worth it” means: the clear cost model of gas makes it easy to define optimality.³

Our main contributions are: (i) an SMT encoding of a subset of EVM bytecode semantics (Section 4), (ii) an implementation of two flavors of superoptimization: *basic*, where the constraint solver is used to check equivalence of enumerated candidate instruction sequences, and *unbounded*, where also the enumeration itself is shifted to the constraint solver (Section 5), and (iii) two large scale evaluations (Section 6). First, we run `ebso` on a collection of smart contracts from a programming competition aimed at producing the cheapest EVM bytecode for given programming challenges. Even in this already highly optimized data set `ebso` still finds 19 optimizations. In the second evaluation we compare the performance of basic and unbounded superoptimization on the 2500 most called smart contracts from the Ethereum blockchain and find that, in our setting, unbounded superoptimization outperforms basic superoptimization.

2 Ethereum and the EVM

Smart contracts in Ethereum are usually written in a specialized high-level language such as Solidity or Vyper and then compiled into *bytecode*, which is executed on the EVM. The EVM is a virtual machine formally defined in the Ethereum yellow paper [14]. It is based on a *stack*, which holds *words*, *i.e.*, bit vectors, of size 256.⁴ The maximal *stack size* is set to 2^{10} . Pushing words onto a full stack leads to a *stack overflow*, while removing words from the empty stack leads to a *stack underflow*. Both lead the EVM to enter an *exceptional halting* state. The EVM also features a volatile *memory*, a word-addressed byte array, and a persistent key-value *storage*, a word-addressed word array, whose contents are stored on the Ethereum blockchain. The bytecode directly corresponds to

³ Of course setting the gas price of individual instructions, such that it accurately reflects the computational cost is hard, and has been a problem in the past see *e.g.* news.ycombinator.com/item?id=12557372.

⁴ This word size was chosen to facilitate the cryptographic computations such as hashing that are often performed in the EVM.

more human-friendly *instructions*. For example, the EVM bytecode 6029600101 encodes the following sequence of instructions: PUSH 41 PUSH 1 ADD. Instructions can be classified into different categories, such as *arithmetic operations*, *e.g.* ADD and SUB for addition and subtraction, *comparisons*, *e.g.* SLT for signed less-than, and *bitwise operations*, like AND and NOT. The instruction PUSH pushes a word onto the stack, while POP removes the top word.⁵ Words on the stack can be duplicated using DUP i and swapped using SWAP i for $1 \leq i \leq 16$, where i refers to the i th word below the top. Some instructions are specific to the blockchain domain, like BLOCKHASH, which returns the hash of a recently mined block, or ADDRESS, which returns the address of the currently executing account. Instructions for control flow include *e.g.* JUMP, JUMPDEST, and STOP.

We write $\delta(\iota)$ for the number of words that instruction ι takes from the stack, and $\alpha(\iota)$ for the number of words ι adds onto the stack. A *program* p is a finite sequence of instructions. We define the *size* $|p|$ of a program as the number of its instructions. To execute a program on the Ethereum blockchain, the caller has to pay *gas*. The amount to be paid depends on both the instructions of the program and the input: every instruction comes with a *gas cost*. For example, PUSH and ADD currently cost 3 g, and therefore executing the program above costs 9 g. Most instructions have a fixed cost, but some take the current state of the execution into account. A prominent example of this behavior is storage. Writing to a zero-valued key conceptually allocates new storage and thus is more expensive than writing to a key that is already in use, *i.e.*, holds a non-zero value. The gas prices of all instructions are specified in the yellow paper [14].

3 Superoptimization

Given a *source* program p superoptimization tries to generate a *target* program p' such that (i) p' is equivalent to p , and (ii) the cost of p' is minimal with respect to a given cost function C . This problem arises in several contexts with different source and target languages. In our case, *i.e.*, for a binary recompiler, both source and target are EVM bytecode.

A standard approach to superoptimization and synthesis [4, 9, 12, 13] is to search through the space of *candidate instruction* sequences of increasing cost and use a constraint solver to check whether a candidate correctly implements the source program. The solver of choice is usually a Satisfiability Modulo Theories (SMT) solver, which operates on first-order formulas in combination with background theories, such as the theory of bit vectors or arrays. Modern SMT solvers are highly optimized and implement techniques to handle arbitrary first-order formulas, such as E-matching. With increasing cost of the candidate sequence, the search space dramatically increases. To deal with this explosion one idea is to hand some of the search to the solver, by using templates [4, 13]. Templates leave holes in the target program, *e.g.* for immediate arguments of instructions, that the solver must then fill. A candidate program is correct if the

⁵ We gloss over the 32 different PUSH instructions depending on the size of the word to be pushed.

<pre> 1: function BASICSo(p_s, C) 2: $n \leftarrow 0$ 3: while true do 4: for all $p_t \in \{p \mid C(p) = n\}$ do 5: $\chi \leftarrow \text{ENCODEBSO}(p_s, p_t)$ 6: if SATISFIABLE(χ) then 7: $m \leftarrow \text{GETMODEL}(\chi)$ 8: $p_t \leftarrow \text{DECODEBSO}(m)$ 9: return p_t 10: $n \leftarrow n + 1$ </pre>	<pre> 1: function UNBOUNDEDSO(p_s, C) 2: $p_t \leftarrow p_s$ 3: $\chi \leftarrow \text{ENCODEUSO}(p_t) \wedge \text{BOUND}(p_t, C)$ 4: while SATISFIABLE(χ) do 5: $m \leftarrow \text{GETMODEL}(\chi)$ 6: $p_t \leftarrow \text{DECODEUSO}(m)$ 7: $\chi \leftarrow \chi \wedge \text{BOUND}(p_t, C)$ 8: return p_t </pre>
(a) Basic Superoptimization.	(b) Unbounded Superoptimization.

Alg. 2: Superoptimization.

encoding is satisfiable, *i.e.*, if the solver finds a model. Constructing the target program then amounts to obtaining the values for the templates from the model. This approach is shown in Algorithm 2(a).

Unbounded superoptimization [5, 6] pushes this idea further. Instead of searching through candidate programs and calling the SMT solver on them, it shifts the search into the solver, *i.e.*, the encoding expresses all candidate instruction sequences of any length that correctly implement the source program. This approach is shown in Algorithm 2(b): if the solver returns satisfiable then there is an instruction sequence that correctly implements the source program. Again, this target program is reconstructed from the model. If successful, a constraint asking for a cheaper program is added and the solver is called again. Note that this also means that unbounded superoptimization can stop with a correct, but possibly non-optimal solution. In contrast, basic superoptimization cannot return a correct solution until it has finished.

The main ingredients of superoptimization in Algorithm 2 are ENCODEBSO/USO producing the SMT encoding, and DECODEBSO/USO reconstructing the target program from a model. We present our encodings for the semantics of EVM bytecode in the following section.

4 Encoding

We start by encoding three parts of the EVM execution state: (i) the stack, (ii) gas consumption, and (iii) whether the execution is in an exceptional halting state. We model the stack as an uninterpreted function together with a counter, which points to the next free position on the stack.

Definition 1. A state $\sigma = \langle \text{st}, \text{c}, \text{hlt}, \text{g} \rangle$ consists of

- (i) a function $\text{st}(\mathcal{V}, j, n)$ that, after the program has executed j instructions on input variables from \mathcal{V} returns the word from position n in the stack,
- (ii) a function $\text{c}(j)$ that returns the number of words on the stack after executing j instructions. Hence $\text{st}(\mathcal{V}, j, \text{c}(j) - 1)$ returns the top of the stack.

- (iii) a function $\text{hlt}(j)$ that returns true (\top) if exceptional halting has occurred after executing j instructions, and false (\perp) otherwise.
- (iv) a function $\mathbf{g}(\mathcal{V}, j)$ that returns the amount of gas consumed after executing j instructions.

Here the functions in σ represent *all* execution states of a program, indexed by variable j .

Example 2. Symbolically executing the program PUSH 41 PUSH 1 ADD using our representation above we have

$$\begin{array}{cccc}
 \mathbf{g}(0) = 0 & \mathbf{g}(1) = 3 & \mathbf{g}(2) = 6 & \mathbf{g}(3) = 9 \\
 \mathbf{c}(0) = 0 & \mathbf{c}(1) = 1 & \mathbf{c}(2) = 2 & \mathbf{c}(3) = 1 \\
 \text{st}(1, 0) = 41 & \text{st}(2, 0) = 41 & \text{st}(2, 1) = 1 & \text{st}(3, 0) = 42
 \end{array}$$

and $\text{hlt}(0) = \text{hlt}(1) = \text{hlt}(2) = \text{hlt}(3) = \perp$.

Note that this program does not consume any words that were already on the stack. This is not the case in general. For instance we might be dealing with the body of a function, which takes its arguments from the stack. Hence we need to ensure that at the beginning of the execution sufficiently many words are on the stack. To this end we first compute the *depth* $\hat{\delta}(p)$ of the program p , *i.e.*, the number of words a program p consumes. Then we take variables $x_0, \dots, x_{\hat{\delta}(p)-1}$ that represent the input to the program and initialize our functions accordingly.

Definition 2. For a program with $\hat{\delta}(p) = d$ we initialize the state σ using

$$\mathbf{g}_\sigma(0) = 0 \wedge \text{hlt}_\sigma(0) = \perp \wedge \mathbf{c}_\sigma(0) = d \wedge \bigwedge_{0 \leq \ell < d} \text{st}_\sigma(\mathcal{V}, 0, \ell) = x_\ell$$

For instance, for the program consisting of the single instruction **ADD** we set $\mathbf{c}(0) = 2$, and $\text{st}(\{x_0, x_1\}, 0, 0) = x_0$ and $\text{st}(\{x_0, x_1\}, 0, 1) = x_1$. We then have $\text{st}(\{x_0, x_1\}, 1, 0) = x_1 + x_2$.

To encode the effect of **EVM** instructions we build SMT formulas to capture their operational semantics. That is, for an instruction ι and a state σ we give a formula $\tau(\iota, \sigma, j)$ that defines the effect on state σ if ι is the j -th instruction that is executed. Since large parts of these formulas are similar for every instruction and only depend on δ and α we build them from smaller building blocks.

Definition 3. For an instruction ι and state σ we define:

$$\begin{aligned}
 \tau_{\mathbf{g}}(\iota, \sigma, j) &\equiv \mathbf{g}_\sigma(\mathcal{V}, j+1) = \mathbf{g}_\sigma(\mathcal{V}, j) + C(\sigma, j, \iota) \\
 \tau_{\mathbf{c}}(\iota, \sigma, j) &\equiv \mathbf{c}_\sigma(j+1) = \mathbf{c}_\sigma(j) + \alpha(\iota) - \delta(\iota) \\
 \tau_{\text{pres}}(\iota, \sigma, j) &\equiv \forall n. n < \mathbf{c}_\sigma(j) - \delta(\iota) \rightarrow \text{st}_\sigma(\mathcal{V}, j+1, n) = \text{st}_\sigma(\mathcal{V}, j, n) \\
 \tau_{\text{hlt}}(\iota, \sigma, j) &\equiv \text{hlt}_\sigma(j+1) = \text{hlt}_\sigma(j) \vee \mathbf{c}_\sigma(j) - \delta(\iota) < 0 \vee \mathbf{c}_\sigma(j) - \delta(\iota) + \alpha(\iota) > 2^{10}
 \end{aligned}$$

Here $C(\sigma, j, \iota)$ is the gas cost of executing instruction ι on state σ after j steps.

The formula τ_g adds the cost of ι to the gas cost incurred so far. The formula τ_c updates the counter for the number of words on the stack according to δ and α . The formula τ_{pres} expresses that all words on the stack below $c_\sigma(j) - \delta(\iota)$ are preserved. Finally, τ_{hit} captures that exceptions relevant to the stack can occur through either an underflow or an overflow, and that once it has occurred an exceptional halt state persists. For now the only other component we need is how the instructions affect the stack st , *i.e.*, a formula $\tau_{\text{st}}(\iota, \sigma, j)$. Here we only give an example and refer to our implementation or the yellow paper [14] for details. We have

$$\begin{aligned} \tau_{\text{st}}(\text{ADD}, \sigma, j) &\equiv \text{st}_\sigma(\mathcal{V}, j+1, c_\sigma(j+1) - 1) \\ &= \text{st}_\sigma(\mathcal{V}, j, c_\sigma(j) - 1) + \text{st}_\sigma(\mathcal{V}, j, c_\sigma(j) - 2) \end{aligned}$$

Finally these formulas yield an encoding for the semantics of an instruction.

Definition 4. *For an instruction ι and state σ we define*

$$\tau(\iota, \sigma, j) \equiv \tau_{\text{st}}(\iota, \sigma, j) \wedge \tau_c(\iota, \sigma, j) \wedge \tau_g(\iota, \sigma, j) \wedge \tau_{\text{hit}}(\iota, \sigma, j) \wedge \tau_{\text{pres}}(\iota, \sigma, j)$$

Then to encode the semantics of a program p all we need to do is to apply τ to the instructions of p .

Definition 5. *For a program $p = \iota_0 \cdots \iota_n$ we set $\tau(p, \sigma) \equiv \bigwedge_{0 \leq j \leq n} \tau(\iota_j, \sigma, j)$.*

Before building an encoding for superoptimization we consider another aspect of the EVM for our state representation: storage and memory. The gas cost for storing words depends on the words that are currently stored. Similarly, the cost for using memory depends on the number of bytes currently used. This is why the cost of an instruction $C(\sigma, j, \iota)$ depends on the state and the function \mathbf{g}_σ accumulating gas cost depends on \mathcal{V} .

To add support for storage and memory to our encoding there are two natural choices: the theory of arrays or an Ackermann encoding. However, since we have not used arrays so far, they would require the solver to deal with an additional theory. For an Ackermann encoding we only need uninterpreted functions, which we have used already. Hence, to represent storage in our encoding we extend states with an uninterpreted function $\mathbf{str}(\mathcal{V}, j, k)$, which returns the word at key k after the program has executed j instructions. Similarly to how we set up the initial stack we need to deal with the values held by the storage before the program is executed. Thus, to initialize \mathbf{str} we introduce fresh variables to represent the initial contents of the storage. More precisely, for all **SLOAD** and **SSTORE** instructions occurring at positions j_1, \dots, j_ℓ in the source program, we introduce fresh variables s_1, \dots, s_ℓ and add them to \mathcal{V} . Then for a state σ we initialize \mathbf{str}_σ by adding the following conjunct to the initialization constraint from Definition 2:

$$\forall w. \mathbf{str}_\sigma(\mathcal{V}, 0, w) = \text{ite}(w = a_{j_1}, s_1, \text{ite}(w = a_{j_2}, s_2, \dots, \text{ite}(w = a_{j_\ell}, s_\ell, w_\perp)))$$

where $a_j = \text{st}_\sigma(\mathcal{V}, j, c(j) - 1)$ and w_\perp is the default value for words in the storage.

The effect of the two storage instructions `SLOAD` and `SSTORE` can then be encoded as follows:

$$\begin{aligned}\tau_{\text{st}}(\text{SLOAD}, \sigma, j) &\equiv \text{st}_\sigma(\mathcal{V}, j+1, \mathbf{c}_\sigma(j+1) - 1) = \text{str}(\mathcal{V}, j, \text{st}_\sigma(\mathcal{V}, j, \mathbf{c}_\sigma(j) - 1)) \\ \tau_{\text{str}}(\text{SSTORE}, \sigma, j) &\equiv \forall w. \text{str}_\sigma(\mathcal{V}, j+1, w) = \\ &\quad \text{ite}(w = \text{st}_\sigma(\mathcal{V}, j, \mathbf{c}_\sigma(j) - 1), \text{st}_\sigma(\mathcal{V}, j, \mathbf{c}_\sigma(j) - 2), \text{str}_\sigma(\mathcal{V}, j, w))\end{aligned}$$

Moreover all instructions except `SSTORE` preserve the storage, that is, for $\iota \neq \text{SSTORE}$ we add the following conjunct to $\tau_{\text{pres}}(\iota, \sigma, j)$:

$$\forall w. \text{str}_\sigma(\mathcal{V}, j+1, w) = \text{str}_\sigma(\mathcal{V}, j, w)$$

To encode memory a similar strategy is an obvious way to go. However, we first want to evaluate the solver's performance on the encodings obtained when using stack and storage. Since the solver already struggled, due to the size of the programs and the number of universally quantified variables, see Section 6, we have not yet added an encoding of memory.

Finally, to use our encoding for superoptimization we need an encoding of equality for two states after a certain number of instructions. Either to ensure that two programs are equivalent (they start and end in equal states) or different (they start in equal states, but end in different ones). The following formula captures this constraint.

Definition 6. For states σ_1 and σ_2 and program locations j_1 and j_2 we define

$$\begin{aligned}\epsilon(\sigma_1, \sigma_2, j_1, j_2) &\equiv \mathbf{c}_{\sigma_1}(j_1) = \mathbf{c}_{\sigma_2}(j_2) \wedge \text{hlt}_{\sigma_1}(j_1) = \text{hlt}_{\sigma_2}(j_2) \\ &\quad \wedge \forall n. n < \mathbf{c}_{\sigma_1}(j_1) \rightarrow \text{st}_{\sigma_1}(\mathcal{V}, j_1, n) = \text{st}_{\sigma_2}(\mathcal{V}, j_2, n) \\ &\quad \wedge \forall w. \text{str}_{\sigma_1}(\mathcal{V}, j_1, w) = \text{str}_{\sigma_2}(\mathcal{V}, j_2, w)\end{aligned}$$

Since we aim to improve gas consumption, we do not demand equality for `g`.

We now have all ingredients needed to implement basic superoptimization: simply enumerate all possible programs ordered by gas cost and use the encodings to check equivalence. However, since already for one `PUSH` there are 2^{256} possible arguments, this will not produce results in a reasonable amount of time. Hence we use templates as described in Section 3. We introduce an uninterpreted function $\mathbf{a}(j)$ that maps a program location j to a word, which will be the argument of `PUSH`. The solver then fills these templates and we can get the values from the model. This is a step forward, but since we have 80 encoded instructions, enumerating all permutations still yields too large a search space. Hence we use an encoding similar to the CEGIS algorithm [4]. Given a collection of instructions, we formulate a constraint representing all possible permutations of these instructions. It is satisfiable if there is a way to connect the instructions into a target program that is equivalent to the source program. The order of the instructions can again be reconstructed from the model provided by the solver. More precisely given a source program p and a list of candidate instructions ι_1, \dots, ι_n , `ENCODEBSO` from Algorithm 2(a) takes variables j_1, \dots, j_n and two states σ and σ' and builds

the following formula

$$\begin{aligned} \forall \mathcal{V}. \epsilon(\sigma, \sigma', 0, 0) \wedge \epsilon(\sigma, \sigma', |p|, n) \wedge \tau(p, \sigma) \\ \wedge \bigwedge_{1 \leq \ell \leq n} \tau(\iota_\ell, \sigma', j_\ell) \wedge \bigwedge_{1 \leq \ell < k \leq n} j_\ell \neq j_k \wedge \bigwedge_{1 \leq \ell \leq n} j_\ell \geq 0 \wedge j_\ell < n \end{aligned}$$

Here the first line encodes the source program, and says that the start and final states of the two programs are equivalent. The second line encodes the effect of the candidate instructions and enforces that they are all used in some order. If this formula is satisfiable we can simply get the j_i from the model and reorder the candidate instructions accordingly to obtain the target program.

Unbounded superoptimization shifts even more of the search into the solver, encoding the search space of all possible programs. To this end we take a variable n , which represents the number of instructions in the target program and an uninterpreted function $\text{instr}(j)$, which acts as a template, returning the instruction to be used at location j . Then, given a set of candidate instructions the formula to encode the search can be built as follows:

Definition 7. *Given a set of instructions Cl we define the formula $\rho(\sigma, n)$ as*

$$\forall j. j \geq 0 \wedge j < n \rightarrow \bigwedge_{\iota \in \text{Cl}} \text{instr}(j) = \iota \rightarrow \tau(\iota, \sigma, j) \wedge \bigvee_{\iota \in \text{Cl}} \text{instr}(j) = \iota$$

Finally, the constraint produced by `ENCODEUSO` from Algorithm 2(b) is

$$\forall \mathcal{V}. \tau(p, \sigma) \wedge \rho(\sigma', n) \wedge \epsilon(\sigma, \sigma', 0, 0) \wedge \epsilon(\sigma, \sigma', |p|, n) \wedge \mathbf{g}_\sigma(\mathcal{V}, |p|) > \mathbf{g}_{\sigma'}(\mathcal{V}, n)$$

During our experiments we observed that the solver struggles to show that the formula is unsatisfiable when p is already optimal. To help in these cases we additionally add a bound on n : since the cheapest EVM instruction has gas cost 1, the target program cannot use more instructions than the gas cost of p , *i.e.*, we add $n \leq \mathbf{g}_\sigma(\mathcal{V}, |p|)$.

In our application domain there are many instructions that fetch information from the outside world. For instance, `ADDRESS` gets the Ethereum address of the account currently executing the bytecode of this smart contract. Since it is not possible to know these values at compile time we cannot encode their full semantics. However, we would still like to take advantage of structural optimizations where these instructions are involved, *e.g.*, via `DUP` and `SWAP`.

Example 3. Consider the program `ADDRESS DUP1`. The same effect can be achieved by simply calling `ADDRESS ADDRESS`. Duplicating words on the stack, if they are used multiple times, is an intuitive approach. However, because executing `ADDRESS` costs $2g$ and `DUP1` costs $3g$, perhaps unexpectedly, the second program is cheaper.

To find such optimizations we need a way to encode `ADDRESS` and similar instructions. For our purposes, these instructions have in common that they put arbitrary but fixed words onto the stack. Analogous to uninterpreted functions, we call them *uninterpreted instructions* and collect them in the set `UI`. To represent

their output we use universally quantified variables—similar to input variables. To encode the effect uninterpreted instructions have on the stack, *i.e.*, τ_{st} , we distinguish between *constant* and *non-constant* uninterpreted instructions.

Let $\text{ui}_c(p)$ be the set of *constant uninterpreted instructions* in p , *i.e.* $\text{ui}_c(p) = \{\iota \in p \mid \iota \in \text{UI} \wedge \delta(\iota) = 0\}$. Then for $\text{ui}_c(p) = \{\iota_1, \dots, \iota_k\}$ we take variables $u_{\iota_1}, \dots, u_{\iota_k}$ and add them to \mathcal{V} , and thus to the arguments of the state function st . The formula τ_{st} can then use these variables to represent the unknown word produced by the uninterpreted instruction, *i.e.*, for $\iota \in \text{ui}_c(p)$ with the corresponding variable u_ι in \mathcal{V} , we set $\tau_{\text{st}}(\iota, \sigma, j) \equiv \text{st}_\sigma(\mathcal{V}, j + 1, \text{c}_\sigma(j)) = u_\iota$.

For a *non-constant instruction* ι , such as **BLOCKHASH** or **BALANCE**, the word put onto the stack by ι depends on the top $\delta(\iota)$ words of the stack. We again model this dependency using an uninterpreted function. That is, for every non-constant uninterpreted instruction ι in the source program p , $\text{ui}_n(p) = \{\iota \in p \mid \iota \in \text{UI} \wedge \delta(\iota) > 0\}$, we use an uninterpreted function f_ι . Conceptually, we can think of f_ι as a read-only memory initialized with the values that the calls to ι produce.

Example 4. The instruction **BLOCKHASH** gets the hash of a given block b . Thus optimizing the program **PUSH** b_1 **BLOCKHASH** **PUSH** b_2 **BLOCKHASH** depends on the values b_1 and b_2 . If $b_1 = b_2$ then the cheaper program **PUSH** b_1 **BLOCKHASH** **DUP1** yields the same state as the original program.

To capture this behaviour, we need to associate the arguments b_1 and b_2 of **BLOCKHASH** with the two different results they may produce. As with constant uninterpreted instructions, to model arbitrary but fixed results, we add fresh variables to \mathcal{V} . However, to account for different results produced by ℓ invocations of ι in p we have to add ℓ variables. Let p be a program and $\iota \in \text{ui}_n(p)$ a unary instruction which appears ℓ times at positions j_1, \dots, j_ℓ in p . For variables u_1, \dots, u_ℓ , we initialize f_ι as follows:

$$\forall w. f_\iota(\mathcal{V}, w) = \text{ite}(w = a_{j_1}, u_1, \text{ite}(w = a_{j_2}, u_2, \dots, \text{ite}(w = a_{j_\ell}, u_\ell, w_\perp)))$$

where a_j is the word on the stack after j instructions in p , that is $a_j = \text{st}_\sigma(\mathcal{V}, j, \text{c}(j) - 1)$, and w_\perp is a default word.

This approach straightforwardly extends to instructions with more than one argument. Here we assume that uninterpreted instructions put exactly one word onto the stack, *i.e.*, $\alpha(\iota) = 1$ for all $\iota \in \text{UI}$. This assumption is easily verified for the EVM: the only instructions with $\alpha(\iota) > 1$ are **DUP** and **SWAP**. Finally we set the effect a non-constant uninterpreted instruction ι with associated function f_ι has on the stack:

$$\tau_{\text{st}}(\iota, \sigma, j) \equiv \text{st}_\sigma(\mathcal{V}, j + 1, \text{c}_\sigma(j + 1) - 1) = f_\iota(\mathcal{V}, \text{st}_\sigma(\mathcal{V}, j, \text{c}_\sigma(j) - 1))$$

For some uninterpreted instructions there might a be way to partially encode their semantics. The instruction **BLOCKHASH** returns 0 if it is called for a block number greater than the current block number. While the current block number is not known at compile time, the instruction **NUMBER** does return it. Encoding this interplay between **BLOCKHASH** and **NUMBER** could potentially be exploited for finding optimizations.

5 Implementation

We implemented basic and unbounded superoptimization in our tool `ebso`, which is available under the Apache-2.0 license: github.com/juliannagele/ebso. The encoding employed by `ebso` uses several background theories: (i) uninterpreted functions (UF) for encoding the state of the EVM, for templates, and for encoding uninterpreted instructions, (ii) bit vector arithmetic (BV) for operations on words, (iii) quantifiers for initial words on the stack and in the storage, and the results of uninterpreted instructions, and (iv) linear integer arithmetic (LIA) for the instruction counter. Hence following the SMT-LIB classification⁶ `ebso`'s constraints fall under the logic UFBVLIA. As SMT solver we chose Z3 [3], version 4.7.1 which we call with default configurations. In particular, Z3 performed well for the theory of quantified bit vectors and uninterpreted functions in the last SMT competition (albeit non-competing).⁷

The aim of our implementation is to provide a prototype without relying on heavy engineering and optimizations such as exploiting parallelism or tweaking Z3 strategies. But without any optimization, for the full word size of the EVM—256 bit—`ebso` did not handle the simple program `PUSH 0 ADD POP` within a reasonable amount of time. Thus we need techniques to make `ebso` viable. By investigating the models generated by Z3 run with the default configuration, we believe that the problem lies with the leading universally quantified variables. And we have plenty of them: for the input on the stack, for the storage, and for uninterpreted instructions. By reducing the word size to a small k , we can reduce the search space for universally quantified variables from 2^{256} to some significantly smaller 2^k . But then we need to check any target program found with a smaller word size.

Example 5. The program `PUSH 0 SUB PUSH 3 ADD` from Example 1 optimizes to `NOT` for word size 2 bit, because then the binary representation of 3 is all ones. When using word size 256 bit this optimization is not correct.

To ensure that the target program has the same semantics for word size 256 bit, we use *translation validation*: we ask the solver to find inputs, which distinguish the source and target programs, *i.e.*, where both programs start in equivalent states, but their final state is different. Using our existing machinery this formula is easy to build:⁸

Definition 8. *Two programs p and p' are equivalent if*

$$\nu(p, p', \sigma, \sigma') \equiv \exists \mathcal{V}, \tau(p, \sigma) \wedge \tau(p', \sigma') \wedge \epsilon(\sigma, \sigma', 0, 0) \wedge \neg \epsilon(\sigma, \sigma', |p|, |p'|)$$

is unsatisfiable. Otherwise, p and p' are different, and the values for the variables in \mathcal{V} from the model are a corresponding witness.

⁶ smtlib.cs.uiowa.edu/logics.shtml

⁷ smt-comp.github.io/2019/results/ufbv-single-query

⁸ This approach also allows for other over-approximations. For instance, we tried using integers instead of bit vectors, which performed worse.

A subtle problem remains: how can we represent the program PUSH 224981 with only k bit? Our solution is to replace arguments a_1, \dots, a_m of PUSH where $a_i \geq 2^k$ with fresh, universally quantified variables c_1, \dots, c_m . If a target program is found, we replace c_i by the original value a_i , and check with translation validation whether this target program is correct. A drawback of this approach is that we might lose potential optimizations.

Example 6. The program PUSH 0b111...111 AND optimizes to the empty program. But, abstracting the argument of PUSH translates the program to PUSH c_i AND, which does not allow the same optimization.

Like many compiler optimizations, `ebso` optimizes basic blocks. Therefore we split EVM bytecode along instructions that change the control flow, *e.g.* JUMPI, or SELFDESTRUCT. Similarly we further split basic blocks into (`ebso`) blocks so that they contain only encoded instructions. Instructions, which are not encoded, or encodable, include instructions that write to memory, *e.g.* MSTORE, or the log instructions LOG.

Lemma 1. *If program p superoptimizes to program t then in any program we can replace p by t .*

Proof. We show the statement by induction on the program context (c_1, c_2) of the program c_1pc_2 . By assumption, the statement holds for the base case $([], [])$. For the step case $(\iota c_1, c_2)$, we observe that every instruction ι is deterministic, *i.e.* executing ι starting from a state σ leads to a deterministic state σ' . By induction hypothesis, executing c_1pc_2 and c_1tc_2 from a state σ' leads to the same state σ'' , and therefore we can replace ιc_1pc_2 by ιc_1tc_2 . We can reason analogously for $(c_1, c_2\iota)$.

6 Evaluation

We evaluated `ebso` on two real-world data sets: (i) optimizing an already highly optimized data set in Section 6.1, and (ii) a large-scale data set from the Ethereum blockchain to compare basic and unboundend superoptimization in Section 6.2. We use `ebso` to extract `ebso` blocks from our data sets. From the extracted blocks (i) we remove duplicate blocks, and (ii) we remove blocks which are only different in the arguments of PUSH by abstracting to word size 4 bit. We run both evaluations on a cluster [7] consisting of nodes running Intel Xeon E5645 processors at 2.40 GHz, with one core and 1 GiB of memory per instance.

We successfully validated all optimizations found by `ebso` by running a reference implementation of the EVM on pseudo-random input. Therefore, we run the bytecode of the original input block and the optimized bytecode to observe that both produce the same final state. The EVM implementation we use is `go-ethereum`⁹ version 1.8.23.

⁹ github.com/ethereum/go-ethereum

	#	%
optimized (optimal)	19 (10)	0.69 % (0.36 %)
proved optimal	481	17.54 %
time-out (trans. val. failed)	2243 (196)	81.77 % (7.15 %)

Table 1: Aggregated results of running `ebso` on `GG`.

6.1 Optimize the Optimized

This evaluation tests `ebso` against human intelligence. Underlying our data set are 200 Solidity contracts (`GGraw`) we collected from the *1st Gas Golfing Contest*.¹⁰ In that contest competitors had to write the most gas-efficient Solidity code for five given challenges: (i) integer sorting, (ii) implementing an interpreter, (iii) hex decoding, (iv) string searching, and (v) removing duplicate elements. Every challenge had two categories: *standard* and *wild*. For *wild*, any Solidity feature is allowed—even inlining EVM bytecode. The winner of each track received 1 Ether. The Gas Golfing Contest provides a very high-quality data set: the EVM bytecode was not only optimized by the `solc` compiler, but also by humans leveraging these compiler optimizations and writing inline code themselves. To collect our data set `GG`, we first compiled the Solidity contracts in `GGraw` with the same set-up as in the contest.¹¹ One contract in the *wild* category failed to compile and was thus excluded from `GGraw`. From the generated `.bin-runtime` files, we extracted our final data set `GG` of 2743 distinct blocks.

For this evaluation, we run `ebso` in its default mode: unbounded superoptimization. We run unbounded superoptimization because, as can be seen in Section 6.2, in our context unbounded superoptimization outperformed basic superoptimization. As time-out for this challenging data set, we estimated 1 h as reasonable.

Table 1 shows the aggregated results of running `ebso` on `GG`. In total, `ebso` optimizes 19 blocks out of 2743, 10 of which are shown to be optimal. Moreover, `ebso` can prove for more than 17 % of blocks in `GG` that they are already optimal. It is encouraging that `ebso` even finds optimizations in this already highly optimized data set. The quality of the data set is supported by the high percentage of blocks being proved as optimal by `ebso`. Next we examine three found optimizations more closely. Our favorite optimization `POP PUSH 1 SWAP1 POP PUSH 0` to `SLT DUP1 EQ PUSH 0` witnesses that superoptimization can find unexpected results, and that unbounded superoptimization can stop with non-optimal results: `SLT DUP1 EQ` is, in fact, a round-about and optimizable way to pop two words from the stack and push 1 on the stack. Some optimizations follow clear patterns. The optimizations `CALLVALUE DUP1 ISZERO PUSH 81` to `CALLVALUE CALLVALUE`

¹⁰ g.solidity.cc

¹¹ Namely, `$ solc --optimize --bin-runtime --optimize-runs 200` with `solc` compiler version 0.4.24 available at github.com/ethereum/solidity/tree/v0.4.24.

	uso		bso		
	#	%	#	%	
optimized (optimal)	943 (393)	1.54 % (0.64 %)	184	0.3 %	
proved optimal	3882	6.34 %	348	0.57 %	
time-out (trans. val. failed)	56 392 (1467)	92.12 % (2.4 %)	60 685	99.13 %	

Table 2: Aggregated results of running `ebso` with `uso` and `bso` on EthBC.

ISZERO PUSH 81 and CALLVALUE DUP1 ISZERO PUSH 364 to CALLVALUE CALLVALUE ISZERO PUSH 364 are both based on the fact that CALLVALUE is cheaper than DUP1. Finding such patterns and generalizing them into peephole optimization rules could be interesting future work.

Unfortunately, `ebso` hit a time-out in nearly 82 % of all cases, where we count a failed translation validation as part of the time-outs, since in that case `ebso` continues to search for optimizations after increasing the word size.

6.2 Unbounded vs. Basic Superoptimization

In this evaluation we compare unbounded and basic superoptimization, which we will abbreviate with `uso` and `bso`, respectively. To compare `uso` and `bso`, we want a considerably larger data set. Fortunately, there is a rich source of EVM bytecode accessible: contracts deployed on the Ethereum blockchain. Assuming that contracts that are called more often are well constructed, we queried the 2500 most called contracts¹² using Google BigQuery.¹³ From them we extract our data set EthBC of 61 217 distinct blocks. For this considerably larger data set, we estimated a cut-off point of 15 min as reasonable. One limitation is that, due to the high volume, we only run the full evaluation once.

Table 2 shows the aggregated results of running `ebso` on EthBC. Out of 61 217 blocks in EthBC, `ebso` finds 943 optimizations using `uso` out of which it proves 393 to be optimal. Using `bso` 184 optimizations are found. Some blocks were shown to be optimal by both approaches. Also, both approaches time out in a majority of the cases: `uso` in more than 92 %, and `bso` in more than 99 %. Over all 61 217 blocks the total amount of gas saved for `uso` is 17 871 and 6903 for `bso`. For all blocks where an optimization is found, the average gas saving per block in `uso` is 29.63 %, and 46.1 % for `bso`. The higher average for `bso` can be explained by (i) `bso`'s bias for smaller blocks, where relative savings are naturally higher, and (ii) `bso` only providing optimal results, whereas `uso` may find intermediate, non-optimal results. The optimization with the largest gain, is one which we did not necessarily expect to find in a deployed contract: a redundant storage access. Storage is expensive, hence optimized for in deployed contracts, but `uso` and

¹² up to block number 7 300 000 deployed on Mar-04-2019 01:22:15 AM +UTC

¹³ cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

`bso` both found `PUSH 0 PUSH 4 SLOAD SUB PUSH 4 DUP2 SWAP1 SSTORE POP` which optimizes to the empty program—because the program basically loads the value from key 4 only to store it back to that same key. This optimization saves at least 5220 g, but up to 20 220 g.

From Table 2 we see that on EthBC, `uso` outperforms `bso` by roughly a factor of five on found optimizations; more than ten times as many blocks are proved optimal by `uso` than by `bso`. As we expected, most optimizations found by `bso` were also found by `uso`, but surprisingly, `bso` found 21 optimizations, on which `uso` failed. We found that nearly all of the 21 source programs are fairly complicated, but have a short optimization of two or three instructions. To pick an example, the block `PUSH 0 PUSH 12 SLOAD LT ISZERO ISZERO ISZERO PUSH 12250` is optimized to the relatively simple `PUSH 1 PUSH 12250`—a candidate block, which will be tried early on in `bso`. Additionally, all 21 blocks are cheap: all cost less than 10 g. We also would have expected at least some of these optimizations to have been found by `uso`. We believe internal unfortunate, non-deterministic choice within the solver to be the reason that it did not.

7 Conclusion

Summary. We develop `ebso`, a superoptimizer for EVM bytecode, implementing two different superoptimization approaches and compare them on a large set of real-world smart contracts. Our experiments show that, relying on the heavily optimized search heuristics of a modern SMT solver is a feasible approach to superoptimizing EVM bytecode.

Related Work. Superoptimization [9] has been explored for a variety of different contexts [5, 6, 10, 12], including binary translation [1] and synthesizing compiler optimizations [11]. To our knowledge `ebso` is the first application of superoptimization to smart contracts.

Chen et al. [2] also aim to save gas by optimizing EVM bytecode. They identified 24 anti patterns by manual inspection. Building on their work we run `ebso` on their identified patterns. For 19 instances, `ebso` too found the same optimizations. For 2 patterns, `ebso` lacks encoding of the instructions (`STOP`, `JUMP`), and for 2 patterns `ebso` times out on a local machine.

Due to the repeated exploitation of flaws in smart contracts, various formal approaches for analyzing EVM bytecode have been proposed. For instance Oyente [8] performs control flow analysis in order to detect security defects such as reentrancy bugs.

Outlook. There is ample opportunity for future work. We do not yet support the EVM’s memory. While conceptually this would be a straightforward extension, the number of universally quantified variables and size of blocks are already posing challenges for performance, as we identified by analyzing the optimizations found by `ebso`.

Thus, it would be interesting to use SMT benchmarks obtained by `ebso`'s superoptimization encoding to evaluate different solvers, *e.g.* `CVC4`¹⁴ or `Vampire`¹⁵. The basis for this is already in place: `ebso` can export the generated constraints in SMT-LIB format. Accordingly, we plan to generate new SMT benchmarks and submit them to one of the suitable categories of SMT-LIB.

In order to ease the burden on developers `ebso` could benefit from caching common optimization patterns [11] to speed up optimization times. Another fruitful approach could be to extract the optimization patterns and generalize them into peephole optimizations and rewrite rules.

References

1. Bansal, S., Aiken, A.: Binary translation using peephole superoptimizers. In: Proc. 8th OSDI. pp. 177–192. USENIX (2008)
2. Chen, T., Li, Z., Zhou, H., Chen, J., Luo, X., Li, X., Zhang, X.: Towards saving money in using smart contracts. In: Proc. 40th ICSE-NIER. pp. 81–84. ACM (2018). <https://doi.org/10.1145/3183399.3183420>
3. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proc. 14th TACAS. LNCS, vol. 9206, pp. 337–340. Springer (2008)
4. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proc. 32nd PLDI. pp. 62–73. ACM (2011). <https://doi.org/10.1145/1993498.1993506>
5. Jangda, A., Yorsh, G.: Unbounded superoptimization. In: Proc. Onward! 2017. pp. 78–88. ACM (2017). <https://doi.org/10.1145/3133850.3133856>
6. Joshi, R., Nelson, G., Randall, K.H.: Denali: A Goal-directed Superoptimizer. In: Proc. 23rd PLDI. pp. 304–314. ACM (2002). <https://doi.org/10.1145/512529.512566>
7. King, T., Butcher, S., Zalewski, L.: Apocrita - High Performance Computing Cluster for Queen Mary University of London (Mar 2017). <https://doi.org/10.5281/zenodo.438045>
8. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proc. 23rd CCS. pp. 254–269. ACM (2016). <https://doi.org/10.1145/2976749.2978309>
9. Massalin, H.: Superoptimizer: A look at the smallest program. In: Proc. 2nd ASPLOS. pp. 122–126. IEEE (1987). <https://doi.org/10.1145/36206.36194>
10. Phothislimthana, P.M., Thakur, A., Bodík, R., Dhurjati, D.: Scaling up superoptimization. In: Proc. 21st ASPLOS. pp. 297–310. ACM (2016). <https://doi.org/10.1145/2872362.2872387>
11. Sasnauskas, R., Chen, Y., Collingbourne, P., Ketema, J., Taneja, J., Regehr, J.: Souper: A synthesizing superoptimizer. CoRR **abs/1711.04422** (2017), <http://arxiv.org/abs/1711.04422>
12. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Proc. 18th ASPLOS. pp. 305–316. ACM (2013). <https://doi.org/10.1145/2451116.2451150>
13. Srinivasan, V., Reps, T.: Synthesis of machine code from semantics. In: Proc. 36th PLDI. pp. 596–607. ACM (2015). <https://doi.org/10.1145/2737924.2737960>
14. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Tech. Rep. Byzantium Version e94ebda (2018), <https://ethereum.github.io/yellowpaper/paper.pdf>

¹⁴ cvc4.cs.stanford.edu/web/

¹⁵ www.vprover.org