Blockchain Superoptimizer

Julian Nagele **Maria A Schett**

2019-10-09 @ LOPSTR @ FM'19

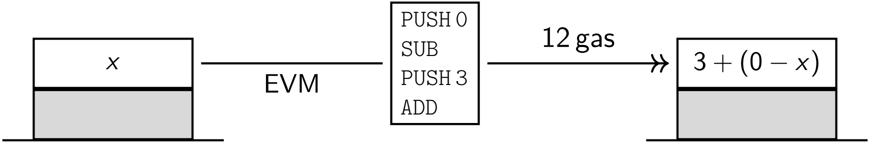
Overview

- Ethereum smart contracts are executed as bytecode on the Ethereum Virtual Machine (EVM)

```
PUSH 0
SUB
PUSH 3
ADD
```

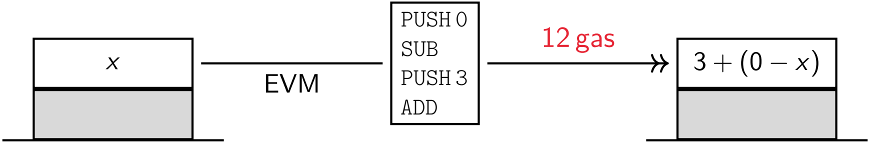
Overview

- Ethereum smart contracts are executed as bytecode on the Ethereum Virtual Machine (EVM)



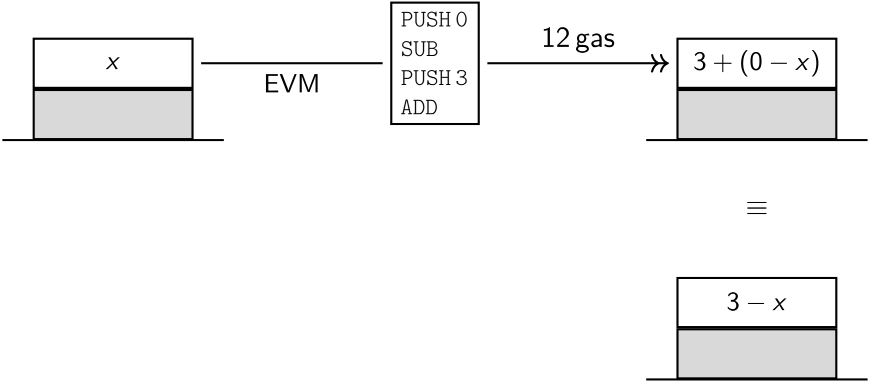
Overview

- Ethereum smart contracts are executed as bytecode on the Ethereum Virtual Machine (EVM)



Overview

- Ethereum smart contracts are executed as bytecode on the Ethereum Virtual Machine (EVM)



x

EVM

```
PUSH 0
SUB
PUSH 3
ADD
```

12 gas

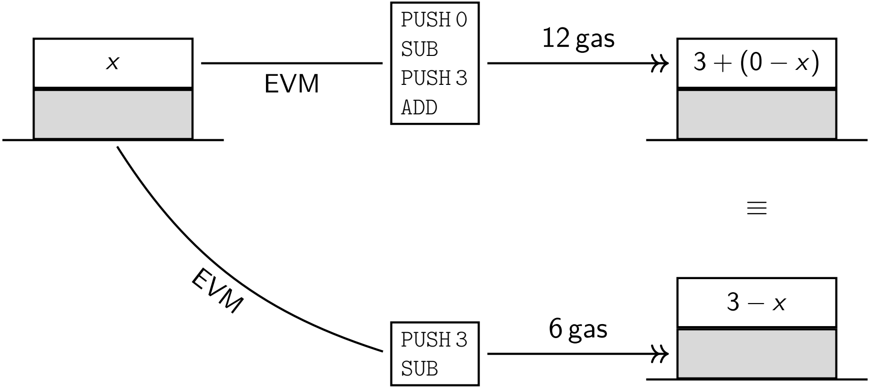
$3 + (0 - x)$

$\equiv$

$3 - x$

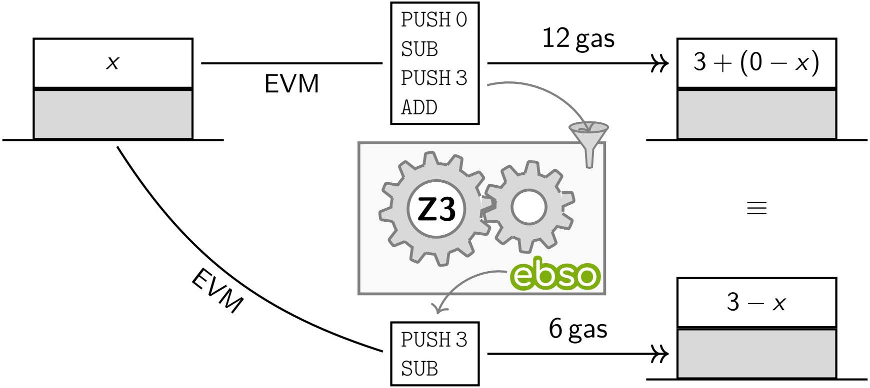
Overview

- Ethereum smart contracts are executed as bytecode on the Ethereum Virtual Machine (EVM)



Overview

- Ethereum smart contracts are executed as bytecode on the Ethereum Virtual Machine (EVM)



```
PUSH 0
SUB
PUSH 3
ADD
```

```
PUSH 3
SUB
```

Goals

- our tool **ebso** superoptimizes EVM bytecode

Goals

- our tool **ebso** superoptimizes EVM bytecode

Ethereum

- EVM gas provides **clear cost model**
- formal semantics [Yellow Paper, 2018]
- $\exists$ data sets for evaluation

Goals

- our tool **ebso** superoptimizes EVM bytecode

Ethereum

- EVM gas provides **clear cost model**
- formal semantics [Yellow Paper, 2018]
- ∃ data sets for evaluation

Today

- feedback on ideas for future work (post-proceedings)

Superoptimization

Superoptimization

- **given:** source program s & cost function C
- **find:** target program t that
  1. has minimal cost $C(t)$
  2. correctly implements s
- **using:** a constraint solver

Superoptimization

- **given:** source program s & cost function C
- **find:** target program t that
  1. has minimal cost $C(t)$
  2. correctly implements s
- **using:** a constraint solver

EVM gas

Superoptimization

- **given:** source program s & cost function C
- **find:** target program t that
  1. has minimal cost $C(t)$
  2. correctly implements s
- **using:** a constraint solver

program $\iota_1 \; \iota_2 \; \cdots \; \iota_n$

Superoptimization

- **given:** source program s & cost function C
- **find:** target program t that
  1. has minimal cost $C(t)$
  2. correctly implements s
- **using:** a constraint solver

$$\begin{aligned} \text{program} \quad & \iota_1 \ \iota_2 \ \cdots \ \iota_n \\ \text{Instr} \quad & \iota \in \{\mathtt{ADD}, \mathtt{SUB}, \mathtt{SLT}, \mathtt{PUSH}\, w, \mathtt{DUP}, \mathtt{SWAP}, \ldots\} \\ & w \in \text{Word} = \{0,1\}^{256} \end{aligned}$$

Superoptimization

- **given:** source program s & cost function C
- **find:** target program t that
  1. has minimal cost $C(t)$
  2. correctly implements s
- **using:** a constraint solver

$$\begin{aligned} \text{program} \quad & \iota_1 \ \iota_2 \ \cdots \ \iota_n \\ \text{Instr} \quad & \iota \in \{\mathtt{ADD}, \mathtt{SUB}, \mathtt{SLT}, \mathtt{PUSH}\,w, \mathtt{DUP}, \mathtt{SWAP}, \ldots\} \\ & w \in \text{Word} = \{0,1\}^{256} \\ & \mathtt{PUSH}\,0\ \ \mathtt{SUB}\ \ \mathtt{PUSH}\,3\ \ \mathtt{ADD} \end{aligned}$$

Superoptimization

- **given:** source program s & cost function C
- **find:** target program t that
  1. has minimal cost $C(t)$
  2. correctly implements s
- **using:** a constraint solver

$$\begin{array}{rl}
\text{program} & \iota_1 \ \iota_2 \ \cdots \ \iota_n \\
\text{Instr} & \iota \in \{\mathtt{ADD}, \mathtt{SUB}, \mathtt{SLT}, \mathtt{PUSH}\, w, \mathtt{DUP}, \mathtt{SWAP}, \ldots\} \\
 & w \in \text{Word} = \{0, 1\}^{256} \\
\text{Pos} & \underline{\mathtt{PUSH}\, 0}_1 \ \underline{\mathtt{SUB}}_2 \ \underline{\mathtt{PUSH}\, 3}_3 \ \underline{\mathtt{ADD}}_4
\end{array}$$

Satisfiability Modulo Theories (SMT) Solver

- first-order logic with background theories
  - bit vectors, integers, uninterpreted functions

Satisfiability Modulo Theories (SMT) Solver

- first-order logic with background theories
  - bit vectors, integers, uninterpreted functions

Uninterpreted Functions

functions a, f, f' s.t.

Satisfiability Modulo Theories (SMT) Solver

- first-order logic with background theories
  - bit vectors, integers, uninterpreted functions

Uninterpreted Functions

functions a, f, f' s.t.

- $a(1) = 3$ $\lambda x.3$

Satisfiability Modulo Theories (SMT) Solver

- first-order logic with background theories
  - bit vectors, integers, uninterpreted functions

Uninterpreted Functions

functions a, f, f' s.t.

- $a(1) = 3$ $\lambda x.3$
- $\forall \ell < 5.\, f(\ell) = f'(\ell)$ $\lambda x.3$

Satisfiability Modulo Theories (SMT) Solver

- first-order logic with background theories
  - bit vectors, integers, uninterpreted functions

Uninterpreted Functions

functions a, f, f' s.t.

- $a(1) = 3$ $\lambda x.3$
- $\forall \ell < 5.\, f(\ell) = f'(\ell)$ $\lambda x.3$
- $f(1) = a(1) \wedge f'(1) = 42$ UNSAT

Superoptimization [Massalin 1987]

- enumerate all possible candidate programs t in increasing cost

$\exists \vec{x}$. to distinguish s & t?

Superoptimization [Massalin 1987]

- enumerate all possible candidate programs *t* in increasing cost

$\exists\vec{x}$. to distinguish *s* & *t*?

- `PUSH` *w* where *w* is a 256 bit word $\implies$ 2^{256} candidates

Superoptimization [Massalin 1987]

- enumerate all possible candidate programs t in increasing cost

$$\exists \vec{x}.\ \text{to distinguish } s \ \& \ t?$$

Templates [Gulwani et al. 2011]

- `PUSH` w where w is a 256 bit word $\implies 2^{256}$ candidates

$$\exists (a : \mathsf{Pos} \to \mathsf{Word})\ \forall \vec{x}.\quad t \text{ implements } s?$$

Superoptimization [Massalin 1987]

- enumerate all possible candidate programs t in increasing cost

$\exists \vec{x}.$ to distinguish s & t?

Templates [Gulwani et al. 2011]

- `PUSH` w where w is a 256 bit word $\implies 2^{256}$ candidates

$\exists (a : \text{Pos} \to \text{Word})\ \forall \vec{x}.\ \ t$ implements s?

Example

$$\texttt{PUSH 0 SUB PUSH 3 ADD} \quad \rightsquigarrow \quad \underline{\texttt{PUSH}\, a(1)}_1\ \underline{\texttt{SUB}}_2$$

$$a(1) = 3 \qquad a(j) = \_$$

Superoptimization [Massalin 1987]

- enumerate all possible candidate programs t in increasing cost

$\exists \vec{x}$. to distinguish s & t?

Templates [Gulwani et al. 2011]

- `PUSH` w where w is a 256 bit word $\implies$ 2^{256} candidates

$\exists(a : \text{Pos} \to \text{Word})\ \forall \vec{x}.\ \ t$ implements s?

Superoptimization [Massalin 1987]

- enumerate all possible candidate programs t in increasing cost

$$\exists \vec{x}.\ \text{to distinguish } s \ \&\ t?$$

Templates [Gulwani et al. 2011]

- $\texttt{PUSH}\, w$ where w is a 256 bit word $\implies$ 2^{256} candidates

$$\exists (a : \text{Pos} \to \text{Word})\ \forall \vec{x}.\quad t \text{ implements } s?$$

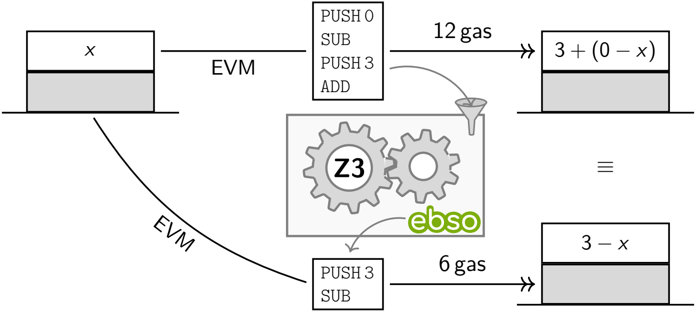
Unbounded Superoptimization [Jangda&Yorsh 2017]

- shift search in solver

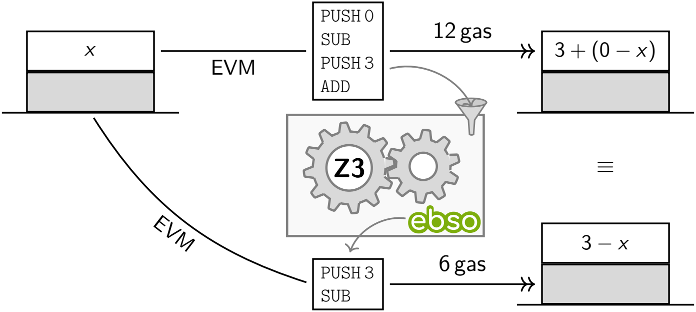
$$\exists (t : \text{Pos} \to \text{Instr})\ \forall \vec{x}.\ t \text{ implements } s \ \&\ C(t) < C(s)?$$

SMT Encoding

Ingredients

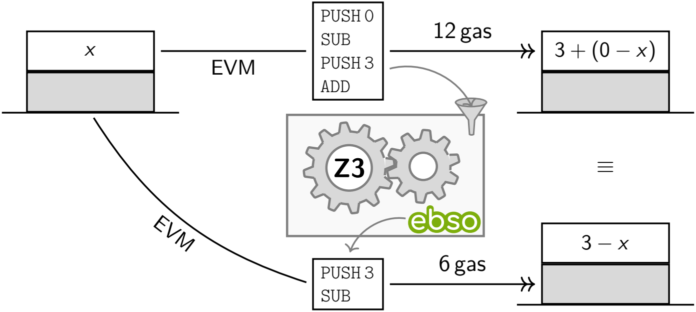


Ingredients



state stack, used gas, stack counter . . .

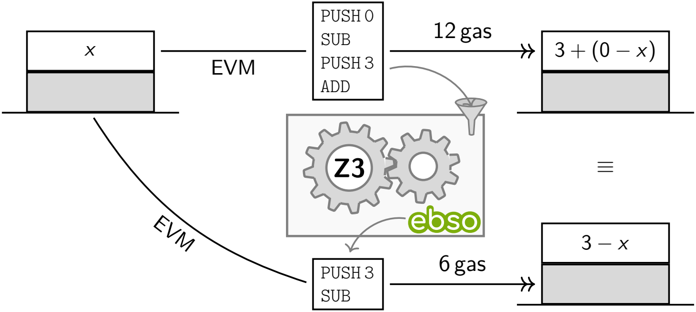
Ingredients



state stack, used gas, stack counter ...

$\twoheadrightarrow$ operational semantics of EVM

Ingredients



state	stack, used gas, stack counter . . .
$\twoheadrightarrow$	operational semantics of EVM
$\equiv$	equality on states

State & $\twoheadrightarrow$

state $\sigma = \langle \mathsf{stk}, \mathsf{c}, \mathsf{g} \rangle$ consists of

- stk(j, ℓ): ℓ-th word on **stack** after j instructions (on input $\vec{x}$)
- g(j): **gas** after j instructions
- **stack counter** $\sim \top_j$ for top of stack after j instructions

State & $\rightarrow\!\!\!\!\rightarrow$

state $\sigma = \langle \mathsf{stk}, \mathsf{c}, \mathsf{g} \rangle$ consists of

- $\mathsf{stk}(j, \ell)$: ℓ-th word on **stack** after j instructions (on input $\vec{x}$)
- $\mathsf{g}(j)$: **gas** after j instructions
- **stack counter** $\sim \top_j$ for top of stack after j instructions

$\sigma_j \xrightarrow{\iota} \sigma_{j+1}$ semantics of $\iota \in \mathsf{Instr}$

State & $\twoheadrightarrow$

state $\sigma = \langle \mathsf{stk}, \mathsf{c}, \mathsf{g} \rangle$ consists of

- $\mathsf{stk}(j, \ell)$: ℓ-th word on **stack** after j instructions (on input $\vec{x}$)
- $\mathsf{g}(j)$: **gas** after j instructions
- **stack counter** $\sim \top_j$ for top of stack after j instructions

$\sigma_j \xrightarrow{\iota} \sigma_{j+1}$ semantics of $\iota \in \mathsf{Instr}$

- $\mathsf{stk}(j+1, \top_{j+1}) = \mathsf{stk}(j, \top_j) +_{bv} \mathsf{stk}(j, \top_j - 1)$ for $\iota =$ `ADD`

State & $\rightarrow\!\!\!\!\rightarrow$

state $\sigma = \langle \mathsf{stk}, \mathsf{c}, \mathsf{g} \rangle$ consists of

- $\mathsf{stk}(j, \ell)$: ℓ-th word on **stack** after j instructions (on input $\vec{x}$)
- $\mathsf{g}(j)$: **gas** after j instructions
- **stack counter** $\sim \top_j$ for top of stack after j instructions

$\sigma_j \xrightarrow{\iota} \sigma_{j+1}$ semantics of $\iota \in \mathsf{Instr}$

- $\mathsf{stk}(j+1, \top_{j+1}) = \mathsf{stk}(j, \top_j) +_{bv} \mathsf{stk}(j, \top_j - 1)$ for $\iota = \mathtt{ADD}$

  $\mathsf{stk}(j+1, \top_{j+1}) = 3$ for $\iota = \mathtt{PUSH}\,3$ and s
  $\mathsf{stk}(j+1, \top_{j+1}) = a(j)$ for $\iota = \mathtt{PUSH}$ and t

State & $\twoheadrightarrow$

state $\sigma = \langle \mathsf{stk}, \mathsf{c}, \mathsf{g} \rangle$ consists of

- $\mathsf{stk}(j, \ell)$: ℓ-th word on **stack** after j instructions (on input $\vec{x}$)
- $\mathsf{g}(j)$: **gas** after j instructions
- **stack counter** $\sim \top_j$ for top of stack after j instructions

$\sigma_j \xrightarrow{\iota} \sigma_{j+1}$ semantics of $\iota \in \mathsf{Instr}$

- $\mathsf{stk}(j+1, \top_{j+1}) = \mathsf{stk}(j, \top_j) +_{bv} \mathsf{stk}(j, \top_j - 1)$ for $\iota = \mathtt{ADD}$

  $\mathsf{stk}(j+1, \top_{j+1}) = u_{\mathtt{ADDRESS}}$ for $\iota = \mathtt{ADDRESS}$

State & $\twoheadrightarrow$

state $\sigma = \langle \mathsf{stk}, \mathsf{c}, \mathsf{g} \rangle$ consists of

- $\mathsf{stk}(j, \ell)$: ℓ-th word on **stack** after j instructions (on input $\vec{x}$)
- $\mathsf{g}(j)$: **gas** after j instructions
- **stack counter** $\sim \top_j$ for top of stack after j instructions

$\sigma_j \xrightarrow{\iota} \sigma_{j+1}$ semantics of $\iota \in \mathsf{Instr}$

- $\mathsf{stk}(j+1, \top_{j+1}) = \mathsf{stk}(j, \top_j) +_{bv} \mathsf{stk}(j, \top_j - 1)$ for $\iota = \texttt{ADD}$

  $\mathsf{stk}(j+1, \top_{j+1}) = \mathsf{storage}_{\texttt{SLOAD}}(\mathsf{stk}(j, \top_j))$ for $\iota = \texttt{SLOAD}$

State & $\rightarrow\!\!\!\rightarrow$

state $\sigma = \langle \mathsf{stk}, \mathsf{c}, \mathsf{g} \rangle$ consists of

- $\mathsf{stk}(j, \ell)$: ℓ-th word on **stack** after j instructions (on input $\vec{x}$)
- $\mathsf{g}(j)$: **gas** after j instructions
- **stack counter** $\sim \top_j$ for top of stack after j instructions

$\sigma_j \overset{\iota}{\to} \sigma_{j+1}$ semantics of $\iota \in \mathsf{Instr}$

- $\mathsf{stk}(j+1, \top_{j+1}) = \mathsf{stk}(j, \top_j) +_{bv} \mathsf{stk}(j, \top_j - 1)$ for $\iota = \mathtt{ADD}$

  & "preserverance of stack" $\wedge$

State & $\twoheadrightarrow$

state $\sigma = \langle \mathsf{stk}, \mathsf{c}, \mathsf{g} \rangle$ consists of

- $\mathsf{stk}(j, \ell)$: ℓ-th word on **stack** after j instructions (on input $\vec{x}$)
- $\mathsf{g}(j)$: **gas** after j instructions
- **stack counter** $\sim \top_j$ for top of stack after j instructions

$\sigma_j \xrightarrow{\iota} \sigma_{j+1}$ semantics of $\iota \in \mathsf{Instr}$

- $\mathsf{stk}(j+1, \top_{j+1}) = \mathsf{stk}(j, \top_j) +_{bv} \mathsf{stk}(j, \top_j - 1)$ for $\iota = \mathtt{ADD}$

  & "preserverance of stack" $\wedge$
- $\mathsf{g}(j+1) = \mathsf{g}(j) + C(\iota) \wedge$

State & $\twoheadrightarrow$

state $\sigma = \langle \mathsf{stk}, \mathsf{c}, \mathsf{g} \rangle$ consists of

- $\mathsf{stk}(j, \ell)$: ℓ-th word on **stack** after j instructions (on input $\vec{x}$)
- $\mathsf{g}(j)$: **gas** after j instructions
- **stack counter** $\sim \top_j$ for top of stack after j instructions

$\sigma_j \xrightarrow{\iota} \sigma_{j+1}$ semantics of $\iota \in \mathsf{Instr}$

- $\mathsf{stk}(j+1, \top_{j+1}) = \mathsf{stk}(j, \top_j) +_{bv} \mathsf{stk}(j, \top_j - 1)$ for $\iota = \mathtt{ADD}$

  & "preserverance of stack" $\wedge$
- $\mathsf{g}(j+1) = \mathsf{g}(j) + C(\iota) \wedge$
- $\top_{j+1} = \top_j + \alpha(\iota) - \delta(\iota)$

State & $\twoheadrightarrow$

state $\sigma = \langle \mathsf{stk}, \mathsf{c}, \mathsf{g} \rangle$ consists of

- $\mathsf{stk}(j, \ell)$: ℓ-th word on **stack** after j instructions (on input $\vec{x}$)
- $\mathsf{g}(j)$: **gas** after j instructions
- **stack counter** $\sim \top_j$ for top of stack after j instructions

$\sigma_j \overset{\iota}{\to} \sigma_{j+1}$ semantics of $\iota \in \mathsf{Instr}$

- $\mathsf{stk}(j+1, \top_{j+1}) = \mathsf{stk}(j, \top_j) +_{bv} \mathsf{stk}(j, \top_j - 1)$ for $\iota = \mathtt{ADD}$

  & "preserverance of stack" $\wedge$
- $\mathsf{g}(j+1) = \mathsf{g}(j) + C(\iota) \wedge$
- $\top_{j+1} = \top_j + \alpha(\iota) - \delta(\iota)$

for program $p = \iota_0, \ldots, \iota_n$, define $\sigma_0 \overset{p}{\twoheadrightarrow} \sigma_{|p|}$ as $\bigwedge_{0 \leqslant j \leqslant n} \sigma_j \overset{\iota_j}{\to} \sigma_{j+1}$

Example

$\underline{\texttt{PUSH}\ a(1)}_1\ \underline{\texttt{SUB}}_2$

Example

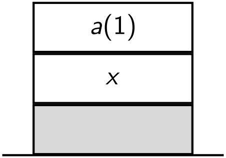
$\underline{\texttt{PUSH}\ a(1)}_1\ \underline{\texttt{SUB}}_2$

$j = 0:$ $\quad \mathsf{stk}(j,1) = x$ $\qquad \mathsf{g}(j) = 0 \quad \top_j = \ 1$

Example

$\underline{\texttt{PUSH}\ a(1)}_1\ \underline{\texttt{SUB}}_2$

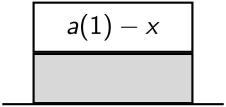
$j = 0:$	$\mathsf{stk}(j,1) = x$		$\mathsf{g}(j) = 0$	$\top_j = 1$
$j = 1:$	$\mathsf{stk}(j,1) = x$	$\mathsf{stk}(j,2) = a(j)$	$\mathsf{g}(j) = 3$	$\top_j = 2$



Example

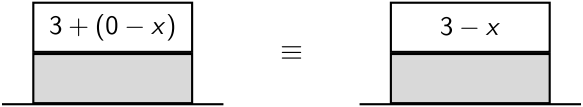
$\underline{\texttt{PUSH}\ a(1)}_1\ \underline{\texttt{SUB}}_2$

$$\begin{array}{llllll}
j = 0: & \mathsf{stk}(j,1) = x & & \mathsf{g}(j) = 0 & \top_j = & 1 \\
j = 1: & \mathsf{stk}(j,1) = x & \mathsf{stk}(j,2) = a(j) & \mathsf{g}(j) = 3 & \top_j = & 2 \\
j = 2: & \mathsf{stk}(j,1) = a(1) - x & & \mathsf{g}(j) = 6 & \top_j = & 1
\end{array}$$



Equality

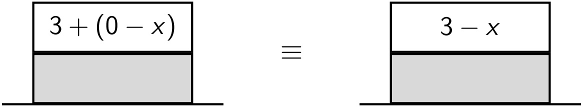
$\sigma_j \equiv \sigma'_{j'}$ states σ and σ' after j and j' instructions



Equality

$\sigma_j \equiv \sigma'_{j'}$ states σ and σ' after j and j' instructions

- $\sigma.\top_j = \sigma'.\top_{j'} \wedge$

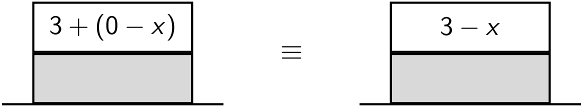


Equality

$\sigma_j \equiv \sigma'_{j'}$ states σ and σ' after j and j' instructions

- $\sigma.\top_j = \sigma'.\top_{j'} \wedge$

  $$\forall \ell < \top_j.\, \sigma.\mathsf{stk}(j, \ell) = \sigma'.\mathsf{stk}(j', \ell)$$



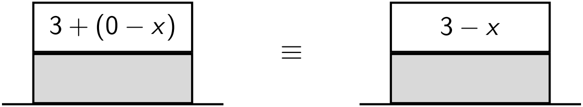
$3 + (0 - x)$ $\equiv$ $3 - x$

Equality

$\sigma_j \equiv \sigma'_{j'}$ states σ and σ' after j and j' instructions

- $\sigma.\top_j = \sigma'.\top_{j'} \wedge$

  $$\forall \ell < \top_j.\, \sigma.\mathsf{stk}(j,\ell) = \sigma'.\mathsf{stk}(j',\ell)$$
- not equal **gas**



Basic Superoptimization

$$\exists (a : \mathrm{Pos} \to \mathrm{Word})\ \forall \vec{x} \text{ s.t. } t \text{ implements } s?$$

Unbounded Superoptimization

$$\exists (t : \mathrm{Pos} \to \mathrm{Instr})\ \forall \vec{x} \text{ s.t. } t \text{ implements } s \;\&\; C(t) < C(s)?$$

Basic Superoptimization

$\exists(a : \text{Pos} \to \text{Word})\ \forall\vec{x}$ s.t. t implements s?

$$\exists a\ \forall\vec{x}.\, \sigma_0 \overset{s}{\twoheadrightarrow} \sigma \wedge \sigma_0 \overset{t}{\twoheadrightarrow} \sigma' \wedge \sigma_{|s|} \equiv \sigma'_{|t|}$$

Unbounded Superoptimization

$\exists(t : \text{Pos} \to \text{Instr})\ \forall\vec{x}$ s.t. t implements s & $C(t) < C(s)$?

Basic Superoptimization

$\exists(a : \mathsf{Pos} \rightarrow \mathsf{Word})\ \forall \vec{x}$ s.t. t implements s?

$$\exists a\ \forall \vec{x}.\, \sigma_0 \overset{s}{\twoheadrightarrow} \sigma \wedge \sigma_0 \overset{t}{\twoheadrightarrow} \sigma' \wedge \sigma_{|s|} \equiv \sigma'_{|t|}$$

Unbounded Superoptimization

$\exists(t : \mathsf{Pos} \rightarrow \mathsf{Instr})\ \forall \vec{x}$ s.t. t implements s & $C(t) < C(s)$?

$$\exists n\ \exists t\ \forall \vec{x}.\, \sigma_0 \overset{s}{\twoheadrightarrow} \sigma \wedge\ \sigma_0 \equiv \sigma'_0 \wedge \sigma_{|s|} \equiv \sigma'_n \wedge \sigma.\mathrm{g}(|s|) > \sigma'.\mathrm{g}(n) \wedge$$

$$\forall j < n.\ \bigwedge_{\iota \in \mathsf{Instr}} t(j) = \iota \implies \sigma'_j \xrightarrow{\iota_j} \sigma'_{j+1} \wedge \bigvee_{\iota \in \mathsf{Instr}} t(j) = \iota \wedge$$

Implementation

Implementation

- available at

`github.com/juliannagele/ebso`

- implemented in OCaml
- $\sim$1.6 kloc (encoding 1 kloc), 635 tests
- using Z3 as SMT solver

Interface

```
$ ./ebso -direct "600003600301"
Optimized PUSH 0 SUB PUSH 3 ADD to
PUSH 3 SUB
Saved 6 gas,
this instruction sequence is optimal.
```

Translation Validation

- **given:** large word size of EVM 256 bit $\implies$ scalability problems
- **solution:** find t for small word size & validate for 256 bit

Translation Validation

- **given:** large word size of EVM 256 bit $\implies$ scalability problems
- **solution:** find t for small word size & validate for 256 bit

$\exists \vec{x}$ s.t. t does not implement s?

$$\exists \vec{x}.\, \sigma_0 \overset{s}{\twoheadrightarrow} \sigma \wedge \sigma_0 \overset{t}{\twoheadrightarrow} \sigma' \wedge \neg(\sigma_{|s|} \equiv \sigma'_{|t|})$$

Translation Validation

- **given:** large word size of EVM 256 bit $\implies$ scalability problems
- **solution:** find t for small word size & validate for 256 bit

$\exists \vec{x}$ s.t. t does not implement s?

$$\exists \vec{x}.\, \sigma_0 \overset{s}{\twoheadrightarrow} \sigma \land \sigma_0 \overset{t}{\twoheadrightarrow} \sigma' \land \neg(\sigma_{|s|} \equiv \sigma'_{|t|})$$

- for word size 2 bit
  - `PUSH 0 SUB PUSH 3 ADD` optimizes to `NOT`
  - because the binary representation of 3 is `11`.

ebso Blocks

- **smart contract**

```
60806040526040805190810160405280600481526020017f48312e3000000000000000000000000
0000000000000000000000000000000008152506006908051906020019062000051929190620001
6e565b503480156200005f57600080fd5b50620186a06000803373fffffffffffffffffffffffff
fffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260
200160002081905550620f424060...
```

`LOG EXP JUMP CALLVALUE DUP1 ISZERO PUSH81 MLOAD`

ebso Blocks

- **smart contract**

60806040526040805190810160405280600481526020017f48312e3000000000000000000000000

000000000000000000000000000000000008152506006908051906020019062000051929190620001

6e565b503480156200005f57600080fd5b50620186a06000803373fffffffffffffffffffffffff

fffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260

2001600020819055550620f424060...

- split at **ebso blocks**

LOG EXP JUMP CALLVALUE DUP1 ISZERO PUSH 81 MLOAD

ebso Blocks

- **smart contract**

```
608060405260408051908101604052806004815260200017f48312e3000000000000000000000000
0000000000000000000000000000000008152506006908051906020019062000051929190620001
6e565b503480156200005f57600080fd5b50620186a06000803373fffffffffffffffffffffffff
fffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260
200160002081905550620f424060...
```

- split at **ebso blocks**
    1. unencoded/unencodable instructions (memory!)

~~LOG~~ ~~EXP~~ JUMP CALLVALUE DUP1 ISZERO PUSH 81 ~~MLOAD~~

ebso Blocks

- **smart contract**

```
60806040526040805190810160405280600481526020017f48312e3000000000000000000000000
0000000000000000000000000000000008152506006908051906020019062000051929190620001
6e565b503480156200005f57600080fd5b50620186a06000803373fffffffffffffffffffffffff
ffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260
200160002081905550620f424060...
```

- split at **ebso blocks**
  1. unencoded/unencodable instructions (memory!)
  2. change in control flow

`~~LOG~~ ~~EXP~~ ~~JUMP~~ CALLVALUE DUP1 ISZERO PUSH 81 ~~MLOAD~~`

Evaluation

EvaluationS

1. **“Optimize the Optimized”**
    - ⋆ Gas Golfing Contest: 199 `Solidity` contracts
    - ⋆ $\implies$ 2743 ebso blocks
2. **“Basic vs. Unbounded”**
    - ⋆ bytecode of 2500 most called contracts from Etherum Blockchain
    - ⋆ $\implies$ 61217 ebso blocks

- 60 min/15 min time-out on 1 core at 2.40 GHz with 1 GiB RAM
- validation with pseudo-random input on `go-ethereum` EVM

Optimize The Optimized (1)

$s = \texttt{CALLVALUE DUP ISZERO PUSH 81}$

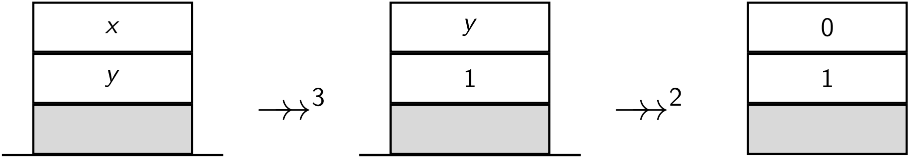
to $t = \texttt{CALLVALUE CALLVALUE ISZERO PUSH 81}$

- $C(\texttt{DUP}) = 3\,\text{gas} > C(\texttt{CALLVALUE}) = 2\,\text{gas}$

Optimize The Optimized (2)

$s = \texttt{POP PUSH 1 SWAP POP PUSH 0}$

to $t = \texttt{SLT DUP EQ PUSH 0}$



Basic vs. Unbounded (1)

- in our setting

Basic vs. Unbounded (1)

- in our setting

$$\textbf{Unbounded} > \textbf{Basic}$$

Basic vs. Unbounded (2)

	Unbounded	
	#	%
optimized		
time-out		
proved optimal		

Basic vs. Unbounded (2)

	Unbounded	
	#	%
optimized[1]	943	1.54 %
time-out		
proved optimal		

---

[1] shown optimal: 393

Basic vs. Unbounded (2)

	Unbounded	
	#	%
optimized[1]	943	1.54 %
time-out	56 392	92.12 %
proved optimal		

---

[1] shown optimal: 393

Basic vs. Unbounded (2)

	Unbounded	
	#	%
optimized[1]	943	1.54 %
time-out	56 392	92.12 %
proved optimal	3882	6.34 %

---

[1] shown optimal: 393

Conclusion

Future Work (1)

- performance/time-outs

Future Work (1)

- performance/time-outs

Feedback

Future Work (1)

- performance/time-outs

Feedback

1. tune encoding
   - not use theory of integers *and* bit vectors, remove storage constraints, . . .

Future Work (1)

- performance/time-outs

Feedback

1. tune encoding
   - ⋆ not use theory of integers *and* bit vectors, remove storage constraints, ...
2. use domain knowledge
   - ⋆ exclude rare instructions from Instr, ...

Future Work (1)

- performance/time-outs

Feedback

1. tune encoding
    - not use theory of integers *and* bit vectors, remove storage constraints, . . .
2. use domain knowledge
    - exclude rare instructions from Instr, . . .
3. tune solver
    - strategy, different solvers

Future Work (2)

- generate peephole optimization rules

Future Work (2)

- generate peephole optimization rules
- SuperOptimization-based Rule Generator **sorg**[2]

---

[2] available at `github.com/mariaschett/sorg`

Future Work (2)

- generate peephole optimization rules
- SuperOptimization-based Rule Generator **sorg**[2]

sorg

- $s =$ `CALLVALUE DUP ISZERO PUSH 81`
  to $t =$ `CALLVALUE CALLVALUE ISZERO PUSH 81`

---

[2] available at `github.com/mariaschett/sorg`

Future Work (2)

- generate peephole optimization rules
- SuperOptimization-based Rule Generator **sorg**[2]

- $s =$ `CALLVALUE DUP ISZERO PUSH 81`
  to $t =$ `CALLVALUE CALLVALUE ISZERO PUSH 81`
- **rule**: `CALLVALUE DUP` → `CALLVALUE CALLVALUE`

---

[2] available at `github.com/mariaschett/sorg`

Future Work (2)

- generate peephole optimization rules
- SuperOptimization-based Rule Generator **sorg**[2]

- $s =$ `CALLVALUE DUP ISZERO PUSH 81` to $t =$ `CALLVALUE CALLVALUE ISZERO PUSH 81`
- **rule**: `CALLVALUE DUP` → `CALLVALUE CALLVALUE`
- found **397** distinct rules from 943 optimized ebso blocks

---

[2] available at `github.com/mariaschett/sorg`

Thank you & questions?

ebso

available at `github.com/juliannagele/ebso`

`[mail@]jnagele.net` `[mail@]maria-a-schett.net`

Bibliography

H. Massalin

Superoptimizer: A Look at the Smallest Program

ASPLOS II, 1987

S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan

Synthesis of Loop-free Programs

Proc. PLDI 2011

A. Jangda and G. Yorsh

Unbounded Superoptimization

Proc. Onward! 2017

Ethereum: A Secure Decentralised Generalised Transaction Ledger

Technical Report Byzantium Version e94ebda