Overview

- Ethereum smart contracts are executed as bytecode on the Ethereum Virtual Machine (EVM).
- Every instruction executed on the EVM has a cost: gas.
- Our tool **ebso** superoptimizes EVM bytecode.

\[
PUSH 0 \\
SUB \\
PUSH 3 \\
ADD \\
12 \text{gas} \\
\]

\[
PUSH 3 \\
SUB \\
6 \text{gas} \\
\]

\[
3 + (0 - x) \\
\equiv \\
3 - x \\
\]
Demo

```
$ ./ebso -translation-validation 256 -direct "600003600301"
Optimized PUSH 0 SUB PUSH 3 ADD to
PUSH 3 SUB
Saved 6 gas, translation validation successful,
this instruction sequence is optimal.
```
Superoptimization

given source program $p_s$ and cost function $C$ find target program $p_t$ that
• correctly implements $p_s$
• has minimal cost

Ethereum
• formal semantics available
• EVM gas provides clear cost model
• large data set readily available for evaluation
• programs are deployed once $\Rightarrow$ long compilation time acceptable

ebso
• does binary recompilation of EVM bytecode
• in two flavors: Basic and Unbounded superoptimization
Basic Superoptimization

1: function BasicSo(p_s, C)
2: n ← 0
3: while true do
4: for all p_t ∈ {p | C(p) = n ∧ checks(p)} do
5:     χ ← EncBso(p_s, p_t)
6:     if Satisfiable(χ) then
7:         m ← GetModel(χ)
8:         p_t ← DecBso(m)
9:     return p_t
10: n ← n + 1

• search through candidate instruction sequences and call solver to check correctness
Unbounded Superoptimization

1: \textbf{function} \textsc{UnboundedSo}(p_s, C) \\
2: \hspace{1em} p_t \leftarrow p_s \\
3: \hspace{1em} \chi \leftarrow \textsc{EncUsO}(p_t) \land \textsc{Bound}(p_t, C) \\
4: \hspace{1em} \textbf{while} \textsc{Satisfiable}(\chi) \hspace{1em} \textbf{do} \\
5: \hspace{2em} m \leftarrow \textsc{GetModel}(\chi) \\
6: \hspace{2em} p_t \leftarrow \textsc{DecUsO}(m) \\
7: \hspace{2em} \chi \leftarrow \chi \land \textsc{Bound}(p_t, C) \\
8: \hspace{1em} \textbf{return} p_t

- shifts the search into the solver
- can stop early with possibly non-optimal solution
Encoding

Satisfiability Modulo Theories

- first-order logic with background theories (bit vectors, integers, uninterpreted functions, arrays, ...)
- powerful off-the-shelf solvers available

Ingredients

- state, i.e., stack, used gas, ...
- $\xrightarrow{p} EVM$, i.e., operational semantics of EVM
- $\equiv$, i.e, equality on states
- for $\text{UNBOUNDED}So$: encoding of search space
State

state $\sigma = \langle \text{st}, c, g \rangle$ consists of

- function $\text{st}(\vec{x}, j, n)$: $n$-th word on stack after $j$ instructions on input $\vec{x}$
- function $c(j)$: number of words on stack after $j$ instructions
- function $g(j)$: amount of gas consumed by first $j$ instructions.

Example

symbolically executing `PUSH 0 SUB PUSH 3 ADD` yields

$g(0) = 0 \quad c(0) = 1 \quad \text{st}(x, 0, 0) = x$

Equality

equality of states after $j_1$ and $j_2 \xrightarrow{\text{EVM}}$ steps

$$\epsilon(\vec{x}, \sigma_1, \sigma_2, j_1, j_2) \equiv c_{\sigma_1}(j_1) = c_{\sigma_2}(j_2)$$

$$\land \forall n < c_{\sigma_1}(j_1). \text{st}_{\sigma_1}(\vec{x}, j_1, n) = \text{st}_{\sigma_2}(\vec{x}, j_2, n)$$
Instructions

semantics of instruction $i$ given by

$$
\tau(i, \vec{x}, \sigma, j) \equiv \tau_g(i, \sigma, j) \land \tau_c(i, \sigma, j) \land \tau_{\text{pres}}(i, \vec{x}, \sigma, j) \land \tau_{\text{st}}(i, \vec{x}, \sigma, j)
$$

$$
\tau_g(i, \sigma, j) \equiv g_\sigma(j + 1) = g_\sigma(j) + C(i)
$$

$$
\tau_c(i, \sigma, j) \equiv c_\sigma(j + 1) = c_\sigma(j) + \alpha(i) - \delta(i)
$$

$$
\tau_{\text{pres}}(i, \vec{x}, \sigma, j) \equiv \forall n < c_\sigma(j) - \delta(i). \text{st}_\sigma(\vec{x}, j, n) = \text{st}_\sigma(\vec{x}, j + 1, n)
$$

$$
\tau_{\text{st}}(\text{ADD}, \vec{x}, \sigma, j) \equiv \text{st}_\sigma(\vec{x}, j, c_\sigma(j + 1) - 1) = \text{st}_\sigma(\vec{x}, j, c_\sigma(j) - 1) +_{bv} \text{st}_\sigma(\vec{x}, j, c_\sigma(j) - 2)
$$

$$
\tau_{\text{st}}(\text{SWAP2}, \vec{x}, \sigma, j) \equiv \text{st}_\sigma(\vec{x}, j, c_\sigma(j + 1) - 1) = \text{st}_\sigma(\vec{x}, j, c_\sigma(j) - 3)
$$

$$
\land \text{st}_\sigma(\vec{x}, j, c_\sigma(j + 1) - 2) = \text{st}_\sigma(\vec{x}, j, c_\sigma(j) - 2)
$$

$$
\land \text{st}_\sigma(\vec{x}, j, c_\sigma(j + 1) - 3) = \text{st}_\sigma(\vec{x}, j, c_\sigma(j) - 1)
$$

for program $p = i_0, \ldots, i_n$ we define

$$
\tau(p, \vec{x}, \sigma) \equiv \bigwedge_{0 \leq j \leq n} \tau(i_j, \vec{x}, \sigma, j)
$$
Superoptimization

Basic

\[ \text{ENCBSO}(\mathcal{p}_s, \mathcal{p}_t) \equiv \forall \vec{x}. \tau(\mathcal{p}_s, \vec{x}, \sigma) \land \tau(\mathcal{p}_t, \vec{x}, \sigma') \land \epsilon(\vec{x}, \sigma, \sigma', 0, 0) \land \epsilon(\vec{x}, \sigma, \sigma', |\mathcal{p}_s|, |\mathcal{p}_t|) \]

Unbounded

\[ \text{ENCUSO}(\mathcal{p}) = \forall \vec{x}. \tau(\mathcal{p}, \vec{x}, \sigma) \land \epsilon(\vec{x}, \sigma, \sigma', 0, 0) \land \epsilon(\vec{x}, \sigma, \sigma', |\mathcal{p}|, n) \land \forall j < n. \bigwedge_{i \in \mathcal{I}} \text{instr}(j) = i \longrightarrow \tau(i, \vec{x}, \sigma', j) \land \bigvee_{i \in \mathcal{I}} \text{instr}(j) = i \]

Templates

- represent subsets of instructions using uninterpreted functions
- for immediate arguments of PUSH use function \( a(j) \) that maps a program location \( j \) to word
- reconstruct actual value from model found by solver
Implementation

- available at github.com/juliannagele/ebso
- implemented in OCaml, using Z3 as SMT solver
- \(\sim 1.6 \text{kloc}\)

Translation Validation

- large word size of EVM (256 bit) led to scalability problems
- solution: find model for small word-size and validate for full size:

\[
\text{TRANSVAL}(p_s, p_t) = \exists \bar{x}. \tau(p_s, \bar{x}, \sigma) \land \tau(p_t, \bar{x}, \sigma') \\
\land \epsilon(\bar{x}, \sigma, \sigma', 0, 0) \land \neg\epsilon(\bar{x}, \sigma, \sigma', |p_s|, |p_t|)
\]
**BasicSo vs. UnboundedSo**

- 11467 Solidity (PL for smart contracts) files from EtherScan platform resulting in 51146 contracts and 89004 encodable sequences of instructions
- 15 min timeout on single core at 2.40 GHz with 1 GiB RAM

<table>
<thead>
<tr>
<th></th>
<th>BasicSo</th>
<th>UnboundedSo</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimized</td>
<td>1214</td>
<td>5407</td>
</tr>
<tr>
<td>proved optimal</td>
<td>2135</td>
<td>17 946</td>
</tr>
<tr>
<td>gas saved</td>
<td>6451</td>
<td>29 973</td>
</tr>
<tr>
<td>weighted gas saved</td>
<td>927 736</td>
<td>2 201 784</td>
</tr>
<tr>
<td>transl. val. failed</td>
<td>n/a</td>
<td>4205</td>
</tr>
</tbody>
</table>
Overlap of BasicSo and UnboundedSo

- overlap between bounded (BSo) and unbounded (USo) superoptimization
- ? indicates USo stopped prematurely

USo?
644
USo
4763
BSo
1032

182
**ebso vs. solc**

- **ebso** on code generated by Solidity compiler with `--optimize` finds

<table>
<thead>
<tr>
<th>optimized</th>
<th>gas saved</th>
<th>weighted gas saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>2609</td>
<td>9325</td>
<td>2,259,871</td>
</tr>
</tbody>
</table>

⇒ optimization potential due to immature compiler
Conclusion

Summary

- **ebso** optimizes EVM bytecode
- unbounded superoptimization shifts search into solver
- relying on search heuristics of solver allows low effort implementation

Future Work

- extend encoding with more EVM features, e.g. storage
- go beyond straight-line code with control flow analysis
- generalize optimization patterns and build into rewrite engine
- extract SMT benchmarks
- develop tactics and strategies to guide SMT solver