Securely Optimized (Ethereum) Smart Contracts using Formal Methods

Elvira Albert

Joint work with: Samir Genaim, Pablo Gordillo, Alejandro Hernández-Cerezo, Enrique Martin Martin and Albert Rubio

Complutense University of Madrid (Spain)

23rd International Conference on Software Engineering and Formal Methods (SEFM'25)

November 12, 2025

Introduction

Introduction

- Ethereum blockchain: enormous growth since first transaction in 2015
- Verification and optimization of Ethereum *smart contracts* raises considerable interest

Introduction

- Ethereum blockchain: enormous growth since first transaction in 2015
- Verification and optimization of Ethereum *smart contracts* raises considerable interest
- Main properties

1. Efficiency: huge volume of transactions, cost and response time increased notably
   - improve scalability to increase capacity
   - optimize execution of smart contracts

Introduction

- Ethereum blockchain: enormous growth since first transaction in 2015
- Verification and optimization of Ethereum *smart contracts* raises considerable interest
- Main properties

1. Efficiency: huge volume of transactions, cost and response time increased notably
    - improve scalability to increase capacity
    - optimize execution of smart contracts
2. Security: smart contracts public and immutable
    - errors exploited by attackers, economic impact
    - formal methods key to ensure security and provide safety guarantees

What do we need to know about smart contracts?

- *Smart contract*: open-source software on the blockchain
  - Not necessarily restricted to the classical concept of contract
  - Collection of secured stored functions
  - All records of the transactions stored on a public and decentralized blockchain

What do we need to know about smart contracts?

- *Smart contract*: open-source software on the blockchain
  - Not necessarily restricted to the classical concept of contract
  - Collection of secured stored functions
  - All records of the transactions stored on a public and decentralized blockchain

- *Ethereum Virtual Machine* is used to run smart contracts
  - Rather standard stack-based language
  - Words of 256-bits
  - Three memory regions
    - ⋆ Operational stack
    - ⋆ Local memory
    - ⋆ Replicated storage

What do we need to know about smart contracts?

- *Smart contract*: open-source software on the blockchain
  - Not necessarily restricted to the classical concept of contract
  - Collection of secured stored functions
  - All records of the transactions stored on a public and decentralized blockchain

- *Ethereum Virtual Machine* is used to run smart contracts
  - Rather standard stack-based language
  - Words of 256-bits
  - Three memory regions
    - ⋆ Operational stack
    - ⋆ Local memory
    - ⋆ Replicated storage
- Relevant features for optimization:
  - Gas-metered execution

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction
- *Price*: gas priced in the cryptocurrencies (*Ether*).

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- *EVM specification:* provides precise definition of gas consumed by each EVM bytecode instruction.

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- *EVM specification:* provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 20.000)

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- *EVM specification:* provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 20.000)

300.000 →

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- *EVM specification:* provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 20.000)

300.000 → | 21.000 + 25.000 execution | →

Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- *EVM specification:* provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 20.000)

300.000 → 21.000 + 25.000 execution → 254.000

Gas-Metered Execution

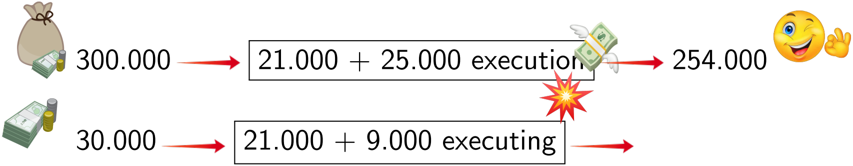
- *Gas-limit*: amount of gas allowed to carry out transaction
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- *EVM specification:* provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 20.000)

300.000 → 21.000 + 25.000 execution → 254.000

30.000 →

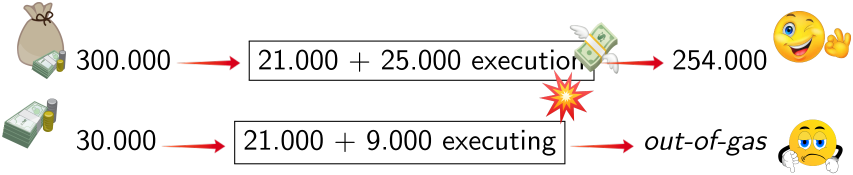
Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- *EVM specification:* provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 20.000)



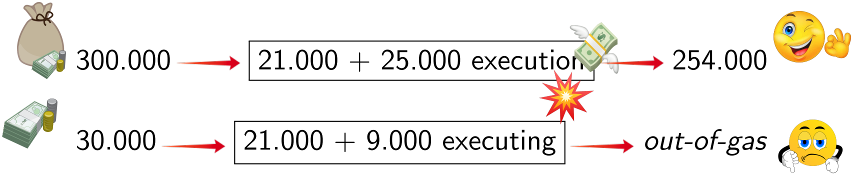
Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- *EVM specification:* provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 20.000)



Gas-Metered Execution

- *Gas-limit*: amount of gas allowed to carry out transaction
- *Price*: gas priced in the cryptocurrencies (*Ether*).
- *EVM specification:* provides precise definition of gas consumed by each EVM bytecode instruction.
- *Gas-model*: instructions that require more computational or storage, cost more (PUSH costs 3 and SSTORE 20.000)



- Rationale of gas metering :
  - prevents attacks based on non-terminating executions;
  - avoids wasting *miners* computational resources;
  - discourages users to overuse replicated *storage*

Superoptimization

Superoptimization is a transformation technique that aims to find the optimal translation of a code by trying all posible loop-free sequences of instructions that produce the same result.

Introduction

Superoptimization

Superoptimization is a transformation technique that aims to find the optimal translation of a code by trying all posible loop-free sequences of instructions that produce the same result.

- **given:** loop-free sequence of code **s** and a cost function **C**
- **find:** target sequence **t** that
  1. has **minimal** cost **C(t)**
  2. correctly implements s
- **using:** constraint solver

Introduction

Superoptimization

Optimization criteria:

- GAS: fee for gas consumption of each EVM bytecode
- SIZE: fee for size in bytes

Introduction

Superoptimization

Optimization criteria:

- GAS: fee for gas consumption of each EVM bytecode
- SIZE: fee for size in bytes

Impact of gas size:

1. reduces the costs of transactions
2. enlarges Ethereum's capability to handle +transactions ...

Introduction

Superoptimization

Optimization criteria:

- GAS: fee for gas consumption of each EVM bytecode
- SIZE: fee for size in bytes

Impact of bytes-size

1. reduces deployment costs
2. maximum bytes-size allowed for deployment

GASOL: GAS Optimization tooLkit

ethereum foundation

Gas and size superoptimization tool for EVM smart contracts

GASOL: GAS Optimization tooLkit

ethereum foundation

Gas and size superoptimization tool for EVM smart contracts

PUSH1 0x17	SWAP1
DUP1	SWAP3
SLOAD	AND
PUSH4 0xffffffff	SWAP2
NOT	SWAP1
AND	SWAP2
PUSH4 0xffffffff	OR
SWAP3	SWAP1

GASOL: GAS Optimization tooLkit

ethereum foundation

Gas and size superoptimization tool for EVM smart contracts

```
PUSH1 0x17
DUP1
SLOAD
PUSH4 0xffffffff
NOT
AND
PUSH4 0xffffffff
SWAP3
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
SWAP1
```

Stack

s0

GASOL: GAS Optimization tooLkit

ethereum foundation

Gas and size superoptimization tool for EVM smart contracts

PUSH1 0x17	SWAP1
DUP1	SWAP3
SLOAD	AND
PUSH4 0xffffffff	SWAP2
NOT	SWAP1
AND	SWAP2
PUSH4 0xffffffff	OR
SWAP3	SWAP1

145 gas

25 bytes

GASOL: GAS Optimization tooLkit

ethereum foundation

Gas and size superoptimization tool for EVM smart contracts

```
PUSH1 0x17
DUP1
SLOAD
PUSH4 0xffffffff
NOT
AND
PUSH4 0xffffffff
SWAP3
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
SWAP1
```

145 gas

25 bytes

Stack

0x17
OR(AND(0xffffffff, s0), AND(NOT(0xffffffff), SLOAD(0x17)))

GASOL: GAS OPTIMIZATION TOOLKIT

ethereum foundation

Gas and size superoptimization tool for EVM smart contracts

```
PUSH1 0x17          SWAP1
DUP1                SWAP3
SLOAD               AND
PUSH4 0xffffffff    SWAP2
NOT                 SWAP1
AND                 SWAP2
PUSH4 0xffffffff    OR
SWAP3               SWAP1
```

145 gas

25 bytes

Stack

0x17
OR(AND(0xffffffff, s0), AND(NOT(0xffffffff), SLOAD(0x17)))

```
PUSH4 0xffffffff
AND
PUSH32 0xff...00
PUSH1 0x17
SLOAD
AND
OR
PUSH1 0x17
```

GASOL: GAS Optimization tooLkit

ethereum foundation

Gas and size superoptimization tool for EVM smart contracts

```
PUSH1 0x17
DUP1
SLOAD
PUSH4 0xffffffff
NOT
AND
PUSH4 0xffffffff
SWAP3
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
SWAP1
```

145 gas
25 bytes

Stack

0x17
OR(AND(0xffffffff, s0), AND(NOT(0xffffffff), SLOAD(0x17)))

121 gas
46 bytes

```
PUSH4 0xffffffff
AND
PUSH32 0xff...00
PUSH1 0x17
SLOAD
AND
OR
PUSH1 0x17
```

GASOL: GAS OPTIMIZATION TOOLKIT

ethereum foundation

Gas and size superoptimization tool for EVM smart contracts

```
PUSH1 0x17          SWAP1
DUP1                SWAP3
SLOAD               AND
PUSH4 0xffffffff    SWAP2
NOT                 SWAP1
AND                 SWAP2
PUSH4 0xffffffff    OR
SWAP3               SWAP1
```

145 gas

25 bytes

127 gas

16 bytes

Stack

0x17
OR(AND(0xffffffff, s0), AND(NOT(0xffffffff), SLOAD(0x17)))

```
PUSH4 0xffffffff
DUP1
NOT
AND
PUSH1 0x17
SLOAD
AND
SWAP2
AND
OR
PUSH1 0x17
```

Plan of the talk

- Part I: Superoptimization framework (CAV'20, PLDI'24)

Plan of the talk

- Part I: Superoptimization framework (CAV'20, PLDI'24)
  - Symbolic execution

Plan of the talk

- Part I: Superoptimization framework (CAV'20, PLDI'24)
  - Symbolic execution
  - Rule-based simplification

Plan of the talk

- Part I: Superoptimization framework (CAV'20, PLDI'24)
  - Symbolic execution
  - Rule-based simplification
  - Optimal synthesis using Max-SMT

Plan of the talk

- Part I: Superoptimization framework (CAV'20, PLDI'24)
  - Symbolic execution
  - Rule-based simplification
  - Optimal synthesis using Max-SMT
- Part II: Extension to memory operations (TACAS'22, ISSTA'24)

Plan of the talk

- Part I: Superoptimization framework (CAV'20, PLDI'24)
  - Symbolic execution
  - Rule-based simplification
  - Optimal synthesis using Max-SMT
- Part II: Extension to memory operations (TACAS'22, ISSTA'24)
- Part III: Neural-guided superoptimization (IST'25)

Plan of the talk

- Part I: Superoptimization framework (CAV'20, PLDI'24)
  - Symbolic execution
  - Rule-based simplification
  - Optimal synthesis using Max-SMT
- Part II: Extension to memory operations (TACAS'22, ISSTA'24)
- Part III: Neural-guided superoptimization (IST'25)
- Part IV: Proof-assistants to verify implementation (CAV'23)

Part I: Basic Superoptimization Framework

PART I: Basic Superoptimization Algorithm

Input: Program $\mathbf{P}$, Objective $\mathbf{Obj}$
Output: Optimized $\mathbf{P}'$, Gain $\mathbf{G}$
Ensures: $\mathbf{P} \equiv \mathbf{P}' \wedge \mathbf{Obj}(\mathbf{P}') \leq \mathbf{Obj}(\mathbf{P})$

PART I: Basic Superoptimization Algorithm

Input: Program **P**, Objective **Obj**
Output: Optimized $\mathbf{P}'$, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P}' \wedge \mathbf{Obj}(\mathbf{P}') \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
```

LoopFreeSequences

Builds the CFG and extracts the sequences from the basic blocks (splits at jumps instructions and store operations)

PART I: Basic Superoptimization Algorithm

Input: Program **P**, Objective **Obj**
Output: Optimized $\mathbf{P'}$, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ←Symbolic(seq, ini)
        fin ←SimplificationRules(final)
```

Symbolic

Generates the symbolic state (initial state and final stack) from a given block and applies simplification rules

PART I: Basic Superoptimization Algorithm

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ←Symbolic(seq, ini)
        fin ←SimplificationRules(final)
        bounds ←ComputeBounds(seq, fin)
```

ComputeBounds

Infers a bound for the number of elements located in the stack and the number of instructions of the solution

PART I: Basic Superoptimization Algorithm

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ←Symbolic(seq, ini)
        fin ←SimplificationRules(final)
        bounds ← ComputeBounds(seq, fin)
        sol, gains ← SearchOptimal(ini, fin, Obj, bounds)
```

SearchOptimal

Finds the best equivalent sequence for the selected objective within the time limit using at Max-SMT solver

PART I: Basic Superoptimization Algorithm

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ←Symbolic(seq, ini)
        fin ←SimplificationRules(final)
        bounds ← ComputeBounds(seq, fin)
        sol, gains ← SearchOptimal(ini, fin, Obj, bounds)
        if gains > 0 ∧ CoqChecker(seq, sol) then
            NewSeq ← NewSeq.append((sol, gains))
        else
            NewSeq ← NewSeq.append((seq, 0))
        end if
    end for
```

CoqChecker

We use the Coq proof-assistant to ensure the optimal solution produces the same symbolic state as the initial state.

PART I: Basic Superoptimization Algorithm

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ← Symbolic(seq, ini)
        fin ← SimplificationRules(final)
        bounds ← ComputeBounds(seq, fin)
        sol, gains ← SearchOptimal(ini, fin, Obj, bounds)
        if gains > 0 ∧ CoqChecker(seq, sol) then
            NewSeq ← NewSeq.append((sol, gains))
        else
            NewSeq ← NewSeq.append((seq, 0))
        end if
    end for
    P′,G ← BuildOptimizedCode(NewSeq)
end procedure
```

BuildOptimizedCode

There is a final step to reconstruct the bytecode from the optimized sequences (e.g., recalculate jump addresses)

From Smart Contracts to Sequences

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \land \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ← Symbolic(seq, ini)
        fin ← SimplificationRules(final)
        bounds ← ComputeBounds(seq, fin)
        sol, gains ← SearchOptimal(ini, fin, Obj, bounds)
        if gains > 0 ∧ CoqChecker(seq, sol) then
            NewSeq ← NewSeq.append((sol, gains))
        else
            NewSeq ← NewSeq.append((seq, 0))
        end if
    end for
    P′,G ← BuildOptimizedCode(NewSeq)
end procedure
```

From Smart Contracts to Sequences

```

pragma solidity 0.8.9;
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
...
contract Bridge is Pool {
    using SafeERC20 for IERC20;

event Send(
    bytes32 transferId,
    ...
);
...
mapping(bytes32 => bool) public transfers;
uint32 public minimalMaxSlippage;
...
function setMinimalMaxSlippage(uint32 _minimalMaxSlippage)
  external onlyGovernor {
    minimalMaxSlippage = _minimalMaxSlippage;
}
...
```

From Smart Contracts to Sequences

...

```
JUMPDEST PUSH1 0x17 DUP1 SLOAD PUSH4 0xffffffff NOT AND PUSH4 0xffffffff
SWAP3 SWAP1 SWAP3 AND SWAP2 SWAP1 SWAP2 OR SWAP1 SSTORE JUMP
JUMPDEST CALLER PUSH1 0x00 SWAP1 DUP2 MSTORE PUSH1 0x8 PUSH1 0x20 MSTORE
PUSH1 0x40 SWAP1 SHA3 SLOAD PUSH1 0xff AND PUSH2 0x15d3 JUMPI
```

...

From Smart Contracts to Sequences

...

JUMPDEST PUSH1 0x17 DUP1 SLOAD PUSH4 0xffffffff NOT AND PUSH4 0xffffffff SWAP3 SWAP1 SWAP3 AND SWAP2 SWAP1 SWAP2 OR SWAP1 SSTORE JUMP JUMPDEST CALLER PUSH1 0x00 SWAP1 DUP2 MSTORE PUSH1 0x8 PUSH1 0x20 MSTORE PUSH1 0x40 SWAP1 SHA3 SLOAD PUSH1 0xff AND PUSH2 0x15d3 JUMPI

...

CFG generator

...

(JUMPDEST PUSH1 0x17 DUP1 SLOAD PUSH4 0xffffffff NOT AND PUSH4 0xffffffff SWAP3 SWAP1 SWAP3 AND SWAP2 SWAP1 SWAP2 OR SWAP1 SSTORE JUMP)$_{B_1}$

(JUMPDEST CALLER PUSH1 0x00 SWAP1 DUP2 MSTORE PUSH1 0x8 PUSH1 0x20 MSTORE PUSH1 0x40 SWAP1 SHA3 SLOAD PUSH1 0xff AND PUSH2 0x15d3 JUMPI)$_{B_2}$

...

From Smart Contracts to Sequences

...

(JUMPDEST PUSH1 0x17 DUP1 SLOAD PUSH4 0xffffffff NOT AND PUSH4 0xffffffff SWAP3 SWAP1 SWAP3 AND SWAP2 SWAP1 SWAP2 OR SWAP1 SSTORE JUMP)$_{B_1}$

(JUMPDEST CALLER PUSH1 0x00 SWAP1 DUP2 MSTORE PUSH1 0x8 PUSH1 0x20 MSTORE PUSH1 0x40 SWAP1 SHA3 SLOAD PUSH1 0xff AND PUSH2 0x15d3 JUMPI)$_{B_2}$

...

Block generation

...

JUMPDEST (PUSH1 0x17 DUP1 SLOAD PUSH4 0xffffffff NOT AND PUSH4 0xffffffff SWAP3 SWAP1 SWAP3 AND SWAP2 SWAP1 SWAP2 OR SWAP1)$_{S_1}$ SSTORE JUMP

JUMPDEST (CALLER PUSH1 0x00 SWAP1 DUP2 MSTORE PUSH1 0x8 PUSH1 0x20)$_{S_2}$ MSTORE

(PUSH1 0x40 SWAP1 SHA3 SLOAD PUSH1 0xff AND PUSH2 0x15d3)$_{S_3}$ JUMPI

...

Symbolic Execution

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \land \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ← Symbolic(seq, ini)
        fin ← SimplificationRules(final)
        bounds ← ComputeBounds(seq, fin)
        sol, gains ← SearchOptimal(ini, fin, Obj, bounds)
        if gains > 0 ∧ CoqChecker(seq, sol) then
            NewSeq ← NewSeq.append((sol, gains))
        else
            NewSeq ← NewSeq.append((seq, 0))
        end if
    end for
    P′,G ← BuildOptimizedCode(NewSeq)
end procedure
```

Symbolic Execution

```
PUSH1 0x17
DUP1
SLOAD
PUSH4 0xffffffff
NOT
AND
PUSH4 0xffffffff
SWAP3
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
SWAP1
```

Symbolic Execution

```
PUSH1 0x17
DUP1
SLOAD
PUSH4 0xffffffff
NOT
AND
PUSH4 0xffffffff
SWAP3
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
SWAP1
```

$\boxed{s_0}$

Symbolic Execution

```
PUSH1 0x17
DUP1
SLOAD
PUSH4 0xffffffff
NOT
AND
PUSH4 0xffffffff
SWAP3
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
SWAP1
```

0x17
s_0

Symbolic Execution

```
  PUSH1 0x17
→ DUP1
  SLOAD
  PUSH4 0xffffffff
  NOT
  AND
  PUSH4 0xffffffff
  SWAP3
  SWAP1
  SWAP3
  AND
  SWAP2
  SWAP1
  SWAP2
  OR
  SWAP1
```

0x17
0x17
s_0

Symbolic Execution

```
  PUSH1 0x17
  DUP1
→ SLOAD
  PUSH4 0xffffffff
  NOT
  AND
  PUSH4 0xffffffff
  SWAP3
  SWAP1
  SWAP3
  AND
  SWAP2
  SWAP1
  SWAP2
  OR
  SWAP1
```

SLOAD(0x17)
0x17
s_0

Symbolic Execution

```
  PUSH1 0x17
  DUP1
  SLOAD
  PUSH4 0xffffffff
  NOT
  AND
→ PUSH4 0xffffffff
  SWAP3
  SWAP1
  SWAP3
  AND
  SWAP2
  SWAP1
  SWAP2
  OR
  SWAP1
```

0xffffffff
AND(NOT(0xffffffff), SLOAD(0x17))
0x17
s_0

Symbolic Execution

```
PUSH1 0x17
DUP1
SLOAD
PUSH4 0xffffffff
NOT
AND
PUSH4 0xffffffff
SWAP3   ←
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
SWAP1
```

s_0
AND(NOT(0xffffffff), SLOAD(0x17))
0x17
0xffffffff

Symbolic Execution

```
PUSH1 0x17
DUP1
SLOAD
PUSH4 0xffffffff
NOT
AND
PUSH4 0xffffffff
SWAP3
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
→SWAP1
```

0x17
OR(AND(0xffffffff,s_0), AND(NOT(0xffffffff), SLOAD(0x17)))

Rule-based Simplification

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ← Symbolic(seq, ini)
        fin ← SimplificationRules(final)
        bounds ← ComputeBounds(seq, fin)
        sol, gains ← SearchOptimal(ini, fin, Obj, bounds)
        if gains > 0 ∧ CoqChecker(seq, sol) then
            NewSeq ← NewSeq.append((sol, gains))
        else
            NewSeq ← NewSeq.append((seq, 0))
        end if
    end for
    P′,G ← BuildOptimizedCode(NewSeq)
end procedure
```

Rule-based Simplification

0x17
OR(AND(0xffffffff, s_0), AND(NOT(0xffffffff), SLOAD(0x17)))

Rule-based Simplification

0x17
OR(AND(0xffffffff,s_0), AND(NOT(0xffffffff), SLOAD(0x17)))

Rule-based Simplification

$$OP(X_{int}) \mapsto eval(OP, X_{int})$$

$$\texttt{NOT(0xffffffff)}$$

0x17
OR(AND(0xffffffff,s_0), AND(NOT(0xffffffff), SLOAD(0x17)))

Rule-based Simplification

$$OP(X_{int}) \mapsto eval(OP, X_{int})$$

`0xffffffffffffffffffffffffffffffffffffffffffffffffffffffff00000000`

0x17
OR(AND(0xffffffff, s_0)), AND(NOT(0xffffffff), SLOAD(0x17)))

Rule-based Simplification

$$OP(X_{int}) \mapsto eval(OP, X_{int})$$

0xffffffffffffffffffffffffffffffffffffffffffffffffffffffff00000000

0x17
OR(AND(0xffffffff,s_0), AND(NOT(0xffffffff), SLOAD(0x17)))

Rule-based Simplification

$$OP(X_{int}) \mapsto eval(OP, X_{int})$$

`0xffffffffffffffffffffffffffffffffffffffffffffffffffffffff00000000`

0x17
OR(AND(0xffffffff,s_0), AND(0xfff...000, SLOAD(0x17)))

Compute Bounds

```
PUSH1 0x17
DUP1
SLOAD
PUSH4 0xffffffff
NOT
AND
PUSH4 0xffffffff
SWAP3
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
SWAP1
```

Bound on the number of instructions of the solution:

Initial length of the block ⇒ **16**

Compute Bounds

```
PUSH1 0x17
DUP1
SLOAD
PUSH4 0xffffffff
NOT
AND
→ PUSH4 0xffffffff
SWAP3
SWAP1
SWAP3
AND
SWAP2
SWAP1
SWAP2
OR
SWAP1
```

Bound on the stack elements:

Max elements stored in the stack ⇒ **4**

0xffffffff
AND(NOT(0xffffffff), SLOAD(0x17))
0x17
s_0

Searching for the Optimal Sequence

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ← Symbolic(seq, ini)
        fin ← SimplificationRules(final)
        bounds ← ComputeBounds(seq, fin)
        sol, gains ← SearchOptimal(ini, fin, Obj, bounds)
        if gains > 0 ∧ CoqChecker(seq, sol) then
            NewSeq ← NewSeq.append((sol, gains))
        else
            NewSeq ← NewSeq.append((seq, 0))
        end if
    end for
    P′,G ← BuildOptimizedCode(NewSeq)
end procedure
```

SMT Encoding Step 1: Flattening

0x17
OR(AND(0xffffffff,s_0), AND(0xff...00, SLOAD(0x17)))

SMT Encoding Step 1: Flattening

$s_1 \mapsto$ `0x17`
$s_2 \mapsto$ `0xffffffff`
$s_3 \mapsto$ `0xff...00`

0x17
OR(AND(0xffffffff, s_0), AND(0xff...00, SLOAD(0x17)))

SMT Encoding Step 1: Flattening

$s_1 \mapsto$ 0x17
$s_2 \mapsto$ 0xffffffff
$s_3 \mapsto$ 0xff...00

s_1
OR(AND(s_2,s_0), AND(s_3,SLOAD(s_1)))

SMT Encoding Step 1: Flattening

$s_1 \mapsto \texttt{0x17}$

$s_2 \mapsto \texttt{0xffffffff}$

$s_3 \mapsto \texttt{0xff...00}$

$s_4 \mapsto \texttt{AND}(s_2, s_0)$

$s_5 \mapsto \texttt{SLOAD}(s_1)$

s_1
$\texttt{OR}(s_4,\texttt{AND}(s_3,s_5))$

SMT Encoding Step 1: Flattening

$s_1 \mapsto \texttt{0x17}$
$s_2 \mapsto \texttt{0xffffffff}$
$s_3 \mapsto \texttt{0xff...00}$
$s_4 \mapsto \texttt{AND}(s_2, s_0)$

$s_5 \mapsto \texttt{SLOAD}(s_1)$
$s_6 \mapsto \texttt{AND}(s_3, s_5)$
$s_7 \mapsto \texttt{OR}(s_4, s_6)$

s_1
s_7

SMT Step 2: Modeling the Stack & Opcodes

`INITIAL`	s_0
`PUSH1 0x17`	$s_1 \mid s_0$
`DUP1`	$s_1 \mid s_1 \mid s_0$
...	...
`SWAP2`	$s_3 \mid s_6 \mid s_2$
`OR`	$s_7 \mid s_1$
`SWAP1`	$s_1 \mid s_7$

SMT Step 2: Modeling the Stack & Opcodes

	0	1	2	3
`INITIAL`	s_0	$\perp$	$\perp$	$\perp$
`PUSH1 0x17`	s_1	s_0	$\perp$	$\perp$
`DUP1`	s_1	s_1	s_0	$\perp$
...	...			
`SWAP2`	s_3	s_6	s_2	$\perp$
`OR`	s_7	s_1	$\perp$	$\perp$
`SWAP1`	s_1	s_7	$\perp$	$\perp$

18

SMT Step 2: Modeling the Stack & Opcodes

$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$
$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$
$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$
...			
$x_{15,0}$	$x_{15,1}$	$x_{15,2}$	$x_{15,3}$
$x_{16,0}$	$x_{16,1}$	$x_{16,2}$	$x_{16,3}$
$x_{17,0}$	$x_{17,1}$	$x_{17,2}$	$x_{17,3}$

18

SMT Step 2: Modeling the Stack & Opcodes

	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$	
t_1	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	
t_2	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	
$\cdots$		$\cdots$			18
t_{15}	$x_{15,0}$	$x_{15,1}$	$x_{15,2}$	$x_{15,3}$	
t_{16}	$x_{16,0}$	$x_{16,1}$	$x_{16,2}$	$x_{16,3}$	
t_{17}	$x_{17,0}$	$x_{17,1}$	$x_{17,2}$	$x_{17,3}$	

SMT Step 2: Modeling the Stack & Opcodes

	$x_{15,0}$	$x_{15,1}$	$x_{15,2}$	$x_{15,3}$
t_{16}	$x_{16,0}$	$x_{16,1}$	$x_{16,2}$	$x_{16,3}$

SMT Step 2: Modeling the Stack & Opcodes

	$x_{15,0}$	$x_{15,1}$	$x_{15,2}$	$x_{15,3}$
t_{16}	$x_{16,0}$	$x_{16,1}$	$x_{16,2}$	$x_{16,3}$

$t_{16} = \texttt{SWAP2}$

SMT Step 2: Modeling the Stack & Opcodes

	$x_{15,0}$	$x_{15,1}$	$x_{15,2}$	$x_{15,3}$
t_{16}	$x_{15,2}$	$x_{16,1}$	$x_{15,0}$	$x_{16,3}$

$t_{16} = \texttt{SWAP2}$

SMT Step 2: Modeling the Stack & Opcodes

	$x_{15,0}$	$x_{15,1}$	$x_{15,2}$	$x_{15,3}$
t_{16}	$x_{15,2}$	$x_{16,1}$	$x_{15,0}$	$x_{16,3}$

$$t_{16} = \texttt{SWAP2} \rightarrow (x_{16,0} = x_{15,2}) \wedge (x_{16,2} = x_{15,0})$$

SMT Step 2: Modeling the Stack & Opcodes

	$x_{15,0}$	$x_{15,1}$	$x_{15,2}$	$x_{15,3}$
t_{16}	$x_{15,2}$	$x_{15,1}$	$x_{15,0}$	$x_{15,3}$

$$t_{16} = \texttt{SWAP2} \rightarrow (x_{16,0} = x_{15,2}) \wedge (x_{16,2} = x_{15,0})$$

SMT Step 2: Modeling the Stack & Opcodes

	$x_{15,0}$	$x_{15,1}$	$x_{15,2}$	$x_{15,3}$
t_{16}	$x_{15,2}$	$x_{15,1}$	$x_{15,0}$	$x_{15,3}$

$$t_{16} = \texttt{SWAP2} \rightarrow (x_{16,0} = x_{15,2}) \wedge (x_{16,2} = x_{15,0}) \wedge (x_{16,1} = x_{15,1}) \wedge (x_{16,3} = x_{15,3})$$

Max-SMT Encoding

- Complete encoding

Max-SMT Encoding

- Complete encoding
  1. Constraints to describe the initial stack

$$I = \bigwedge_{0 \leqslant \alpha < k} (x_{0,\alpha} = s_\alpha)$$

Max-SMT Encoding

- Complete encoding
  1. Constraints to describe the initial stack

$$I = \bigwedge_{0 \leqslant \alpha < k} (x_{0,\alpha} = s_\alpha)$$

  2. Constraints to describe the target stack (previous flattening)

$$F = \bigwedge_{0 \leqslant \alpha < w} (x_{Lbound,\alpha} = f_\alpha)$$

Max-SMT Encoding

- Complete encoding
  1. Constraints to describe the initial stack
$$I = \bigwedge_{0 \leqslant \alpha < k} (x_{0,\alpha} = s_\alpha)$$
  2. Constraints to describe the target stack (previous flattening)
$$F = \bigwedge_{0 \leqslant \alpha < w} (x_{Lbound,\alpha} = f_\alpha)$$
  3. Constraints to describe effect of instructions on the stack

Max-SMT Encoding

- Complete encoding
  1. Constraints to describe the initial stack
  $$I = \bigwedge_{0 \leqslant \alpha < k} (x_{0,\alpha} = s_\alpha)$$
  2. Constraints to describe the target stack (previous flattening)
  $$F = \bigwedge_{0 \leqslant \alpha < w} (x_{Lbound,\alpha} = f_\alpha)$$
  3. Constraints to describe effect of instructions on the stack
  4. Non-stack operations considered as uninterpreted functions

$$hard = \bigwedge_{0 \leqslant j < Lbound} 0 \leq t_j \leq Ins \ \wedge I \ \wedge F \ \wedge C_{\texttt{PUSH}}(j) \ \wedge \ C_{\texttt{SWAP}k}(j) \ \wedge .. \bigwedge_{f \in Ins_U} C_U(j, f$$

Max-SMT Encoding

- Complete encoding
  1. Constraints to describe the initial stack
  $$I = \bigwedge_{0 \leqslant \alpha < k} (x_{0,\alpha} = s_\alpha)$$
  2. Constraints to describe the target stack (previous flattening)
  $$F = \bigwedge_{0 \leqslant \alpha < w} (x_{Lbound,\alpha} = f_\alpha)$$
  3. Constraints to describe effect of instructions on the stack
  4. Non-stack operations considered as uninterpreted functions

  $$hard = \bigwedge_{0 \leqslant j < Lbound} 0 \leq t_j \leq Ins \ \wedge I \ \wedge F \ \wedge C_{\text{PUSH}}(j) \ \wedge \ C_{\text{SWAP}k}(j) \ \wedge .. \bigwedge_{f \in Ins_U} C_U(j, f$$
- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction

Max-SMT Encoding

- Complete encoding
  1. Constraints to describe the initial stack
  $$I = \bigwedge_{0 \leqslant \alpha < k} (x_{0,\alpha} = s_\alpha)$$
  2. Constraints to describe the target stack (previous flattening)
  $$F = \bigwedge_{0 \leqslant \alpha < w} (x_{Lbound,\alpha} = f_\alpha)$$
  3. Constraints to describe effect of instructions on the stack
  4. Non-stack operations considered as uninterpreted functions

$$hard = \bigwedge_{0 \leqslant j < Lbound} 0 \leq t_j \leq Ins \wedge I \wedge F \wedge C_{\mathrm{PUSH}}(j) \wedge C_{\mathtt{SWAP}k}(j) \wedge .. \bigwedge_{f \in Ins_U} C_U(j, f$$

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction
  - SMT formula ⇒ hard constraints

Max-SMT Encoding

- Complete encoding
  1. Constraints to describe the initial stack
     $$I = \bigwedge_{0 \leqslant \alpha < k} (x_{0,\alpha} = s_\alpha)$$
  2. Constraints to describe the target stack (previous flattening)
     $$F = \bigwedge_{0 \leqslant \alpha < w} (x_{Lbound,\alpha} = f_\alpha)$$
  3. Constraints to describe effect of instructions on the stack
  4. Non-stack operations considered as uninterpreted functions

$$hard = \bigwedge_{0 \leqslant j < Lbound} 0 \leq t_j \leq Ins \ \wedge I \ \wedge F \ \wedge C_{\text{PUSH}}(j) \ \wedge \ C_{\texttt{SWAP}k}(j) \ \wedge .. \bigwedge_{f \in Ins_U} C_U(j, f$$

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction
  - SMT formula ⇒ hard constraints
  - Cost of every EVM instruction ⇒ soft constraints

Max-SMT Encoding

- Complete encoding
  1. Constraints to describe the initial stack

$$I = \bigwedge_{0 \leqslant \alpha < k} (x_{0,\alpha} = s_\alpha)$$

  2. Constraints to describe the target stack (previous flattening)

$$F = \bigwedge_{0 \leqslant \alpha < w} (x_{Lbound,\alpha} = f_\alpha)$$

  3. Constraints to describe effect of instructions on the stack
  4. Non-stack operations considered as uninterpreted functions

$$hard = \bigwedge_{0 \leqslant j < Lbound} 0 \leq t_j \leq Ins \ \wedge I \ \wedge F \ \wedge C_{\text{PUSH}}(j) \ \wedge \ C_{\text{SWAP}k}(j) \ \wedge .. \bigwedge_{f \in Ins_U} C_U(j, f$$

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction
  - SMT formula ⇒ hard constraints
  - Cost of every EVM instruction ⇒ soft constraints
  - Two different criteria to optimize (original 145 gas and 25 bytes)

Max-SMT Encoding

- Complete encoding
  1. Constraints to describe the initial stack
  $$I = \bigwedge_{0 \leqslant \alpha < k} (x_{0,\alpha} = s_\alpha)$$
  2. Constraints to describe the target stack (previous flattening)
  $$F = \bigwedge_{0 \leqslant \alpha < w} (x_{Lbound,\alpha} = f_\alpha)$$
  3. Constraints to describe effect of instructions on the stack
  4. Non-stack operations considered as uninterpreted functions

$$hard = \bigwedge_{0 \leqslant j < Lbound} 0 \leq t_j \leq Ins \ \wedge I \ \wedge F \ \wedge C_{\text{PUSH}}(j) \ \wedge \ C_{\text{SWAP}k}(j) \ \wedge .. \bigwedge_{f \in Ins_U} C_U(j, f$$

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction
  - SMT formula ⇒ hard constraints
  - Cost of every EVM instruction ⇒ soft constraints
  - Two different criteria to optimize (original 145 gas and 25 bytes)
    - ⋆ Gas model: 121 gas and 46 bytes

Max-SMT Encoding

- Complete encoding
  1. Constraints to describe the initial stack
     $$I = \bigwedge_{0 \leqslant \alpha < k} (x_{0,\alpha} = s_\alpha)$$
  2. Constraints to describe the target stack (previous flattening)
     $$F = \bigwedge_{0 \leqslant \alpha < w} (x_{Lbound,\alpha} = f_\alpha)$$
  3. Constraints to describe effect of instructions on the stack
  4. Non-stack operations considered as uninterpreted functions

$$hard = \bigwedge_{0 \leqslant j < Lbound} 0 \leq t_j \leq Ins \wedge I \wedge F \wedge C_{\text{PUSH}}(j) \wedge C_{\text{SWAP}k}(j) \wedge .. \bigwedge_{f \in Ins_U} C_U(j, f$$

- Optimization
  - The cost of the solution can be expressed in terms of the cost of every single instruction
  - SMT formula ⇒ hard constraints
  - Cost of every EVM instruction ⇒ soft constraints
  - Two different criteria to optimize (original 145 gas and 25 bytes)
    - Gas model: 121 gas and 46 bytes
    - Bytes-size model:127 gas and 16 bytes

Experimental Evaluation

- Dataset: 30 most recent smart contracts compiled using version 8 of solc, whose source code was available as of June 21, 2021

Experimental Evaluation

- Dataset: 30 most recent smart contracts compiled using version 8 of solc, whose source code was available as of June 21, 2021
- Compiled using solc version 0.8.9 with the optimization flag enabled. **Already optimized code!**

Experimental Evaluation

- Dataset: 30 most recent smart contracts compiled using version 8 of solc, whose source code was available as of June 21, 2021
- Compiled using solc version 0.8.9 with the optimization flag enabled. **Already optimized code!**
- Split in memory and store operations

Experimental Evaluation

- Dataset: 30 most recent smart contracts compiled using version 8 of solc, whose source code was available as of June 21, 2021
- Compiled using solc version 0.8.9 with the optimization flag enabled. **Already optimized code!**
- Split in memory and store operations
- Two objective functions: gas and byte size

Experimental Evaluation

- Dataset: 30 most recent smart contracts compiled using version 8 of solc, whose source code was available as of June 21, 2021
- Compiled using solc version 0.8.9 with the optimization flag enabled. **Already optimized code!**
- Split in memory and store operations
- Two objective functions: gas and byte size
- The primary findings are:

Experimental Evaluation

- Dataset: 30 most recent smart contracts compiled using version 8 of solc, whose source code was available as of June 21, 2021
- Compiled using solc version 0.8.9 with the optimization flag enabled. **Already optimized code!**
- Split in memory and store operations
- Two objective functions: gas and byte size
- The primary findings are:
  1. We obtained size gains and gas gains for the 14% and 28% of the analyzed blocks resp.

Experimental Evaluation

- Dataset: 30 most recent smart contracts compiled using version 8 of solc, whose source code was available as of June 21, 2021
- Compiled using solc version 0.8.9 with the optimization flag enabled. **Already optimized code!**
- Split in memory and store operations
- Two objective functions: gas and byte size
- The primary findings are:
  1. We obtained size gains and gas gains for the 14% and 28% of the analyzed blocks resp.
  2. Bytes optimized: 2.64%

Experimental Evaluation

- Dataset: 30 most recent smart contracts compiled using version 8 of solc, whose source code was available as of June 21, 2021
- Compiled using solc version 0.8.9 with the optimization flag enabled. **Already optimized code!**
- Split in memory and store operations
- Two objective functions: gas and byte size
- The primary findings are:
  1. We obtained size gains and gas gains for the 14% and 28% of the analyzed blocks resp.
  2. Bytes optimized: 2.64%
  3. Gas optimized: 0.64%

Experimental Evaluation

- Dataset: 30 most recent smart contracts compiled using version 8 of solc, whose source code was available as of June 21, 2021
- Compiled using solc version 0.8.9 with the optimization flag enabled. **Already optimized code!**
- Split in memory and store operations
- Two objective functions: gas and byte size
- The primary findings are:
  1. We obtained size gains and gas gains for the 14% and 28% of the analyzed blocks resp.
  2. Bytes optimized: 2.64%
  3. Gas optimized: 0.64%
     - ⋆ ⇒ 167.440,98 USD in one day
     - ⋆ ⇒ more than 61M USD in one year (assuming they are uniformly distributed)

Part II: Extension to Memory Operations

Motivation

- Our basic framework can potentially achieve more savings by including operations beyond stack operations

Motivation

- Our basic framework can potentially achieve more savings by including operations beyond stack operations
- Idea: extend the basic framework to handle *memory* operations

Motivation

- Our basic framework can potentially achieve more savings by including operations beyond stack operations
- Idea: extend the basic framework to handle *memory* operations
- Limitation: great overhead when modelling *memory* extensions in superoptimization
  - In fact, EBSO and SOUPER superoptimizers avoid it

Motivation

- Our basic framework can potentially achieve more savings by including operations beyond stack operations
- Idea: extend the basic framework to handle *memory* operations
- Limitation: great overhead when modelling *memory* extensions in superoptimization
  - In fact, EBSO and SOUPER superoptimizers avoid it
- Proposal: leverage our basic two-staged method
  - Detecting redundant memory accesses as simplification rules
  - Extend the Max-SMT encoding using a *dependency order* among instructions

Superoptimization Algorithm With Memory

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure Superoptimization(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ←Symbolic(seq, ini)
        fin ←SimplificationRules(final)
        bounds ← ComputeBounds(seq, fin)
        sol, gains ← SearchOptimal(ini, fin, Obj, bounds)
        if gains > 0 ∧ CoqChecker(seq, sol) then
            NewSeq ← NewSeq.append((sol, gains))
        else
            NewSeq ← NewSeq.append((seq, 0))
        end if
    end for
    P′,G ← BuildOptimizedCode(NewSeq)
end procedure
```

Superoptimization Algorithm With Memory

Input: Program **P**, Objective **Obj**

Output: Optimized **P′**, Gain **G**

Ensures: $\mathbf{P} \equiv \mathbf{P}' \wedge \mathbf{Obj}(\mathbf{P}') \leq \mathbf{Obj}(\mathbf{P})$

```
procedure Superoptimization(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        final ←Symbolic(seq, ini)
        fin ←SimplificationRules(final)
        bounds ← ComputeBounds(seq, fin)
        sol, gains ← SearchOptimal(ini, fin, Obj, bounds)
        if gains > 0 ∧ CoqChecker(seq, sol) then
            NewSeq ← NewSeq.append((sol, gains))
        else
            NewSeq ← NewSeq.append((seq, 0))
        end if
    end for
    P′,G ← BuildOptimizedCode(NewSeq)
end procedure
```

Symbolic Execution

```
PUSH1 0x80
PUSH1 0x40
MSTORE
...
SLOAD
...
SSTORE
...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
ISZERO
```

Stack:

Memory:

Storage:

Symbolic Execution

```
PUSH1 0x80
PUSH1 0x40
MSTORE
...
SLOAD
...
SSTORE
...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
ISZERO
```

Stack:

Memory:

$\texttt{MSTORE}_3(\texttt{0x80,0x40})$

Storage:

Symbolic Execution

```
PUSH1 0x80
PUSH1 0x40
MSTORE
...
▶SLOAD
...
SSTORE
...
SLOAD
...
SSTORE
...
CALLVALUE
DUP1
ISZERO
```

Stack:

Memory:

$\texttt{MSTORE}_3(\texttt{0x80,0x40})$

Storage:

$\texttt{SLOAD}_{10}(\texttt{0x01})$

Symbolic Execution

```
PUSH1 0x80      SLOAD
PUSH1 0x40      ...
MSTORE          SSTORE
...             ...
SLOAD           CALLVALUE
...             DUP1
SSTORE        ▶ ISZERO
...
```

Stack:

CALLVALUE
ISZ(CALLVALUE)

Memory:

$\mathtt{MSTORE}_3(\mathtt{0x80,0x40})$

Storage:

$\mathtt{SLOAD}_{10}(\mathtt{0x01})$	$\mathtt{SSTORE}_{23}(\mathtt{0x01},V1)$	$\mathtt{SLOAD}_{31}(\mathtt{0x01})$	$\mathtt{SSTORE}_{44}(\mathtt{0x01},V2)$

Rule-based Memory Simplifications

Notion of Conflict

Two memory instructions A and B have a *conflict*, denoted as conf(A,B) if:

(i) A is a store and B is a load and the positions they access might be the same;

(ii) A and B are both stores, the positions they modify might be the same, and they store different values.

Rule-based Memory Simplifications

Notion of Conflict

Two memory instructions A and B have a *conflict*, denoted as conf(A,B) if:

(i) A is a store and B is a load and the positions they access might be the same;

(ii) A and B are both stores, the positions they modify might be the same, and they store different values.

We have the following simplification rules:

Rule-based Memory Simplifications

Notion of Conflict

Two memory instructions A and B have a *conflict*, denoted as conf(A,B) if:

(i) A is a store and B is a load and the positions they access might be the same;

(ii) A and B are both stores, the positions they modify might be the same, and they store different values.

We have the following simplification rules:

- If a `STORE`(p, v) is followed by a `LOAD`(p) instruction with no conflicting `STORE` in between, replace the `LOAD`(p) with v

Rule-based Memory Simplifications

Notion of Conflict

Two memory instructions A and B have a *conflict*, denoted as conf(A,B) if:

(i) A is a store and B is a load and the positions they access might be the same;

(ii) A and B are both stores, the positions they modify might be the same, and they store different values.

We have the following simplification rules:

- If a `STORE`(p, v) is followed by a `LOAD`(p) instruction with no conflicting `STORE` in between, replace the `LOAD`(p) with v
- If two `STORE` instructions access the same position with no `LOAD` in between, remove the first `STORE`

Rule-based Memory Simplifications

Notion of Conflict

Two memory instructions A and B have a *conflict*, denoted as conf(A,B) if:

(i) A is a store and B is a load and the positions they access might be the same;

(ii) A and B are both stores, the positions they modify might be the same, and they store different values.

We have the following simplification rules:

- If a $\texttt{STORE}(p, v)$ is followed by a $\texttt{LOAD}(p)$ instruction with no conflicting `STORE` in between, replace the $\texttt{LOAD}(p)$ with v
- If two `STORE` instructions access the same position with no `LOAD` in between, remove the first `STORE`
- If there is a $\texttt{STORE}(p, \texttt{LOAD}(p))$ instruction and position p is not modified by another `STORE` instruction, remove this instruction

Memory Simplifications - Example

$\texttt{SLOAD}_{10}(\texttt{0x01})$	$\texttt{SSTORE}_{23}(\texttt{0x01},V1)$	$\texttt{SLOAD}_{31}(\texttt{0x01})$	$\texttt{SSTORE}_{44}(\texttt{0x01},V2)$

Memory Simplifications - Example

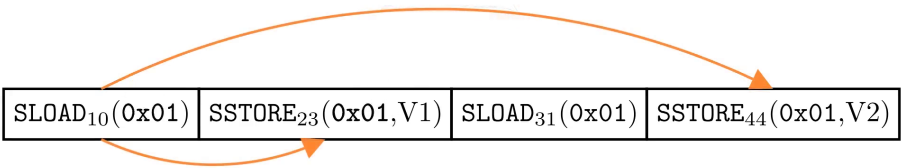
$\texttt{SLOAD}_{10}(\texttt{0x01})$	$\texttt{SSTORE}_{23}(\texttt{0x01},V1)$	$\texttt{SLOAD}_{31}(\texttt{0x01})$	$\texttt{SSTORE}_{44}(\texttt{0x01},V2)$

Memory Simplifications - Example

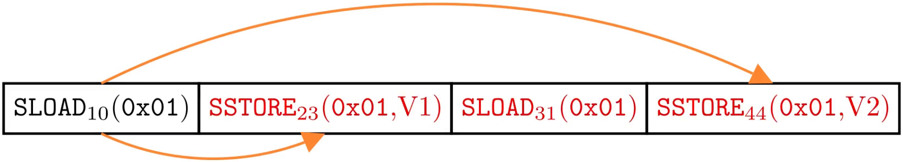
0x01 0x01

$\texttt{SLOAD}_{10}(\texttt{0x01})$	$\texttt{SSTORE}_{23}(\texttt{0x01,V1})$	$\texttt{SLOAD}_{31}(\texttt{0x01})$	$\texttt{SSTORE}_{44}(\texttt{0x01,V2})$

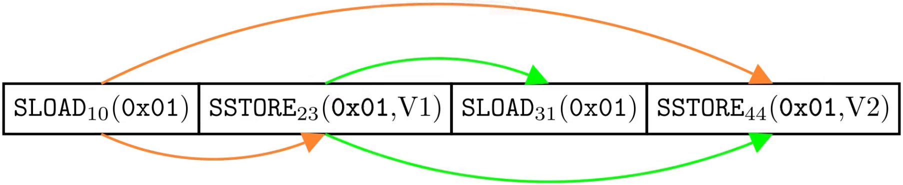
Memory Simplifications - Example



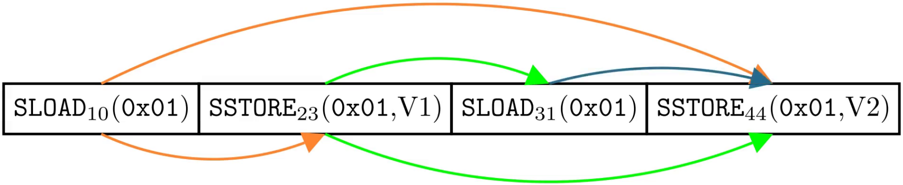
Memory Simplifications - Example



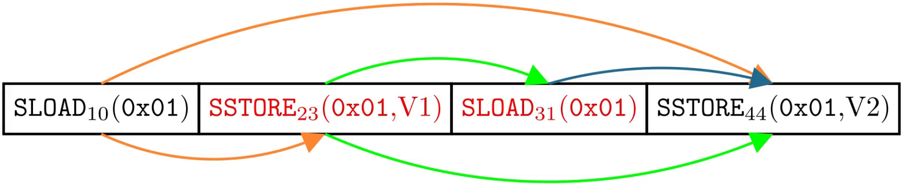
Memory Simplifications - Example



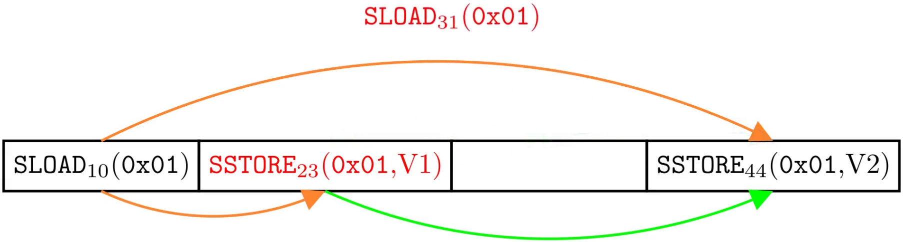
Memory Simplifications - Example



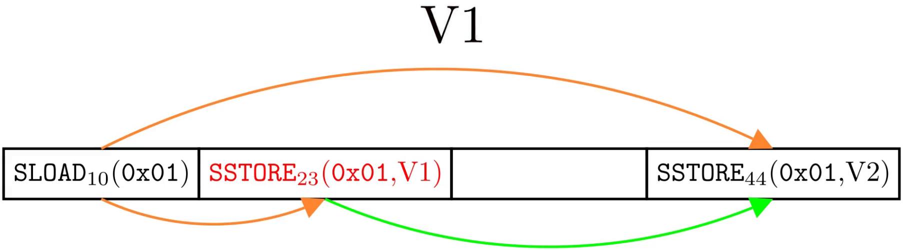
Memory Simplifications - Example



Memory Simplifications - Example

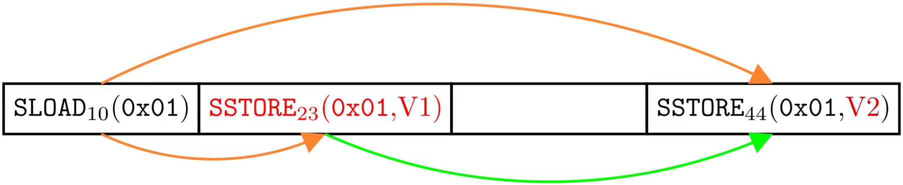


Memory Simplifications - Example

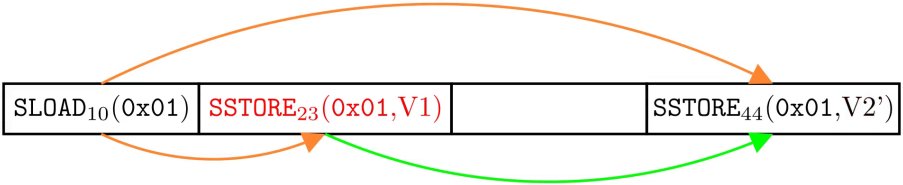


Memory Simplifications - Example

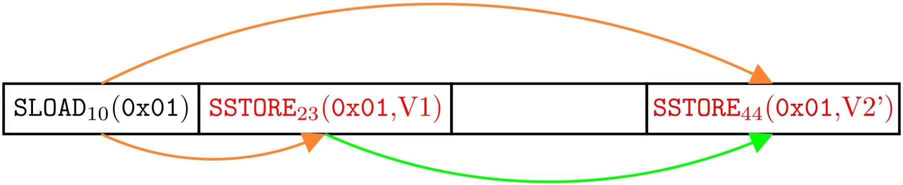
$$V2' = V2[\mathtt{SLOAD}_{31} \mapsto V1]$$



Memory Simplifications - Example



Memory Simplifications - Example



Memory Simplifications - Example

Dependency Order

Dependency Order

Let A and B be two instructions in a sequence S. We say that B has to be executed after A in S, denoted as $A \sqsubset B$ if conf(A,B).

i) There is no conflict between LOADS

ii) Dependencies are added after simplification

Dependency Order

Dependency Order

Let A and B be two instructions in a sequence S. We say that B has to be executed after A in S, denoted as $A \sqsubset B$ if conf(A,B).

i) There is no conflict between LOADS

ii) Dependencies are added after simplification

⇒Introduce variables l_A, l_B to track their positions

$$l_A < l_B \text{ where } A \sqsubset B$$

$$l_{10} < l_{44} \text{ where } \texttt{SLOAD}_{10} \sqsubset \texttt{SSTORE}_{44}$$

Experimental Evaluation

- 1 benchmark set:
  - 30 latest verified smart contracts compiled with latest **solc** version at the moment (0.8.9)

Experimental Evaluation

- 1 benchmark set:
  - 30 latest verified smart contracts compiled with latest **solc** version at the moment (0.8.9)
- The primary findings are:
  1. 0.72% extra gas savings from the code optimized by **solc** (+0.1% from basic version)

Experimental Evaluation

- 1 benchmark set:
  - 30 latest verified smart contracts compiled with latest **solc** version at the moment (0.8.9)
- The primary findings are:
  1. 0.72% extra gas savings from the code optimized by **solc** (+0.1% from basic version)
  2. Gas savings breakdown:
     - ⋆ 14.6% memory rules
     - ⋆ 34.4% other rules
     - ⋆ 51% Max-SMT encoding

Experimental Evaluation

- 1 benchmark set:
  - 30 latest verified smart contracts compiled with latest **solc** version at the moment (0.8.9)
- The primary findings are:
  1. 0.72% extra gas savings from the code optimized by **solc** (+0.1% from basic version)
  2. Gas savings breakdown:
     - ★ 14.6% memory rules
     - ★ 34.4% other rules
     - ★ 51% Max-SMT encoding
  3. 0.72%⇒ 185,754.82 USD in one day
  4. 0.1% ⇒ 26,162.65 USD in one day

Part III: Neural-Guided Superoptimization

Motivation

- The previous extension results in larger blocks to analyze

Motivation

- The previous extension results in larger blocks to analyze
  - Pros: there are more potential optimization gains
  - Cons: scalability poses a greater threat
    - The search space grow exponentially with the length of the sequence

Motivation

- The previous extension results in larger blocks to analyze
  - Pros: there are more potential optimization gains
  - Cons: scalability poses a greater threat
    - The search space grow exponentially with the length of the sequence
- Motivation
  - Quite often the blocks to superoptimize are already optimal
  - The initial bound is often larger than needed, resulting in an unnecessary overhead

Motivation

- The previous extension results in larger blocks to analyze
  - Pros: there are more potential optimization gains
  - Cons: scalability poses a greater threat
    - The search space grow exponentially with the length of the sequence
- Motivation
  - Quite often the blocks to superoptimize are already optimal
  - The initial bound is often larger than needed, resulting in an unnecessary overhead
- Proposal: incorporate machine learning techniques to tackle the previous issues
  - Our superoptimization framework provides us with an unlimited source of information for the supervised learning
  - There are patterns of code identify the previous behaviour

Neural-Guided Superoptimization Algorithm

Input: Program **P**, Objective **Obj**
Output: Optimized $\mathbf{P'}$, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        optimize ← PredictOptimizable(s,Obj)
        if optimize then
```

PredictOptimizable

`PredictOptimizable` classifier that predicts two classes: *already optimal* or *not optimal*

Neural-Guided Superoptimization Algorithm

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure SUPEROPTIMIZATION(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        optimize ←PredictOptimizable(s,Obj)
        if optimize then
            final ← Symbolic(seq, ini)
            fin ← SimplificationRules(final)
            bounds ← ComputeBounds(seq,fin)
```

ComputeBounds

ComputeBounds determines an **upper bound** and a **lower bound** on the minimum number of instructions

Neural-Guided Superoptimization Algorithm

Input: Program **P**, Objective **Obj**
Output: Optimized **P′**, Gain **G**
Ensures: $\mathbf{P} \equiv \mathbf{P'} \wedge \mathbf{Obj}(\mathbf{P'}) \leq \mathbf{Obj}(\mathbf{P})$

```
procedure Superoptimization(P,Obj)
    Seqs ← LoopFreeSequences(P)
    NewSq ←[ ]
    for (seq, ini) ∈ Seqs do
        optimize ←PredictOptimizable(s,Obj)
        if optimize then
            final ← Symbolic(seq, ini)
            fin ← SimplificationRules(final)
            bounds ← ComputeBounds(seq,fin)
            sz ← PredictBlockSizeBound(seq, bounds,Obj)
            sol, gains ← SearchOptimal(ini, fin, Obj, sz, bounds)
            if gains > 0 ∧ CoqChecker(seq, sol) then
                NewSeq ← NewSeq.append((sol, gains))
            else
                NewSeq ← NewSeq.append((seq, 0))
            end if
        end if
    end for
    P′,G ← BuildOptimizedCode(NewSeq)
end procedure
```

PredictBlockSizeBound

PredictBlockSizeBound regression model to predict the length needed to compute an optimal sequence

Experimental Setup

- Training:

Experimental Setup

- Training:
  - Blocks extracted from the last 5,000 verified smart contracts downloaded in three different dates

Experimental Setup

- Training:
  - Blocks extracted from the last 5,000 verified smart contracts downloaded in three different dates
  - Avoid repeated blocks, selecting a representative for each bytecode input representation

Experimental Setup

- Training:
  - Blocks extracted from the last 5,000 verified smart contracts downloaded in three different dates
  - Avoid repeated blocks, selecting a representative for each bytecode input representation
  - 80% for training and 20% for validation

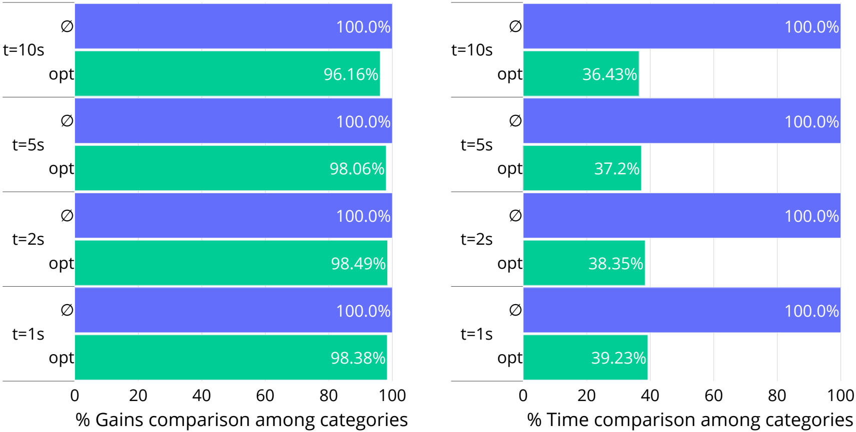
Experimental Setup

- Training:
  - Blocks extracted from the last 5,000 verified smart contracts downloaded in three different dates
  - Avoid repeated blocks, selecting a representative for each bytecode input representation
  - 80% for training and 20% for validation
- Experimental Setup:
  - 100 most-called contracts deployed on Ethereum compiled with recent versions of solc
    - 41,106,276 transactions in total

Experimental Setup

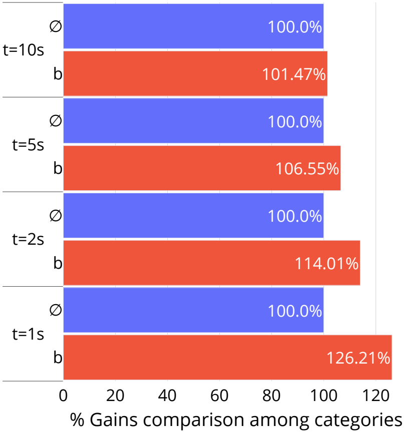
- Training:
  - Blocks extracted from the last 5,000 verified smart contracts downloaded in three different dates
  - Avoid repeated blocks, selecting a representative for each bytecode input representation
  - 80% for training and 20% for validation
- Experimental Setup:
  - 100 most-called contracts deployed on Ethereum compiled with recent versions of solc
    - 41,106,276 transactions in total
  - We run different configurations that with combinations of:
    - Different timeouts $n = 10, 5, 2, 1s$
    - Selectively enabling each of the previous two models

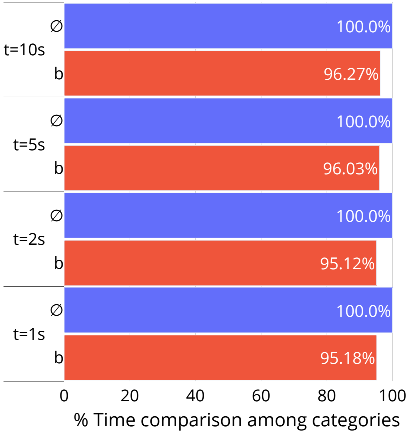
Experimental Evaluation



- `PredictOptimizable`:
  - Preserves more than 96% of the savings!
  - Time reduction up to 40% of the initial configuration

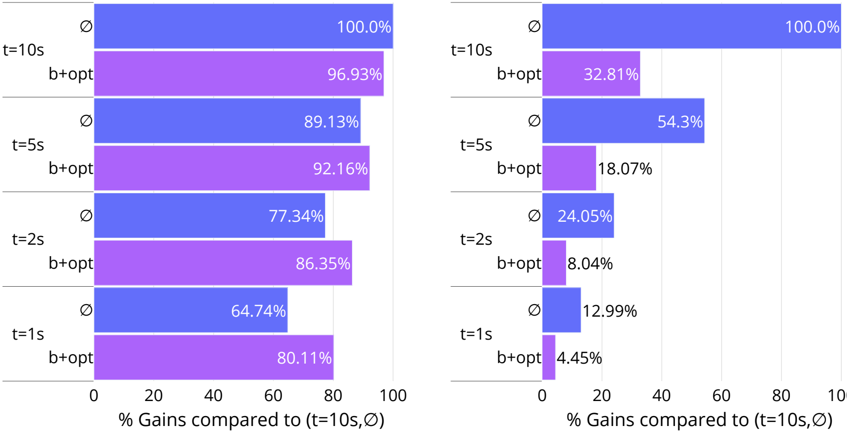
Experimental Evaluation





- `PredictBlockSizeBound`:
  - Achieves $>$ 100% of the gains consistently
  - Greater percentage of gains when decreasing the time limit

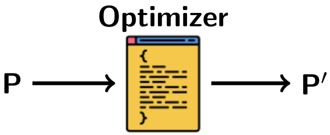
Experimental Evaluation



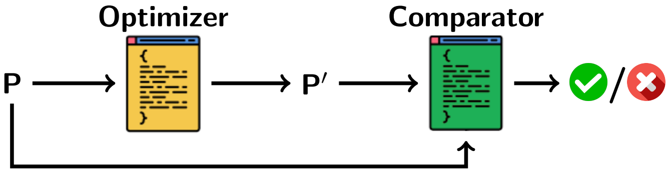
- Configuration chosen: $(t = 2s, \texttt{b+opt})$ with both models enabled
  - Average optimization time per contract: $\sim$3 min
  - $1.24 M savings on these contracts!

Part IV: Verification of Optimization Results

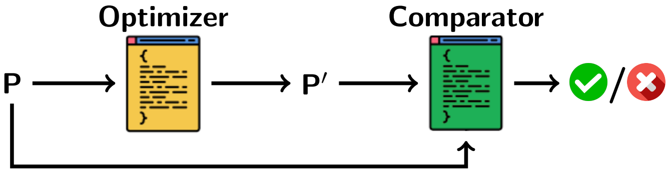
Verification of Results



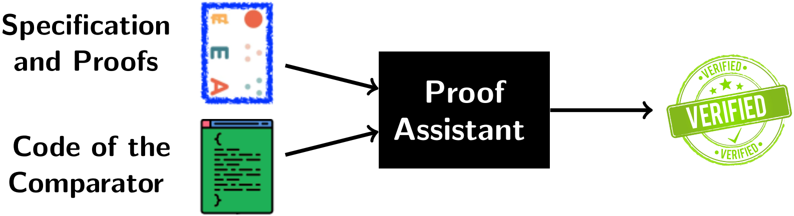
Verification of Results



Verification of Results



We develop the comparator using the proof assistant's (programming) language, and prove its correctness.



Spec: If it returns ✓, then P and P' are semantically equivalent.

Optimization Example

```
PUSH1 0x17, DUP1, SLOAD, PUSH4 0xffffffff, NOT, AND, PUSH4
0xffffffff, SWAP3, SWAP1, SWAP3, AND, SWAP2, SWAP1, SWAP2,OR, SWAP1
```

```
PUSH4 0xffffffff, AND PUSH32, 0xff...00000000, PUSH1 0x17, SLOAD,
AND, OR, PUSH1 0x17
```

Optimization Example

```
PUSH1 0x17, DUP1, SLOAD, PUSH4 0xffffffff, NOT, AND, PUSH4
0xffffffff, SWAP3, SWAP1, SWAP3, AND, SWAP2, SWAP1, SWAP2,OR, SWAP1
```

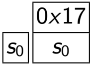
s_0

```
PUSH4 0xffffffff, AND PUSH32, 0xff...00000000, PUSH1 0x17, SLOAD,
AND, OR, PUSH1 0x17
```

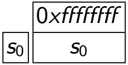
s_0

OPTIMIZATION EXAMPLE

`PUSH1 0x17, DUP1, SLOAD, PUSH4 0xffffffff, NOT, AND, PUSH4 0xffffffff, SWAP3, SWAP1, SWAP3, AND, SWAP2, SWAP1, SWAP2,OR, SWAP1`

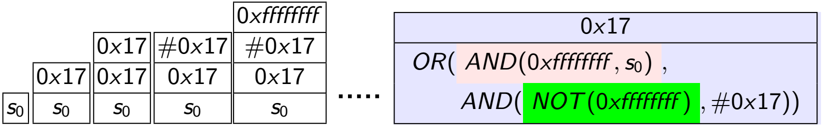


`PUSH4 0xffffffff, AND PUSH32, 0xff...00000000, PUSH1 0x17, SLOAD, AND, OR, PUSH1 0x17`

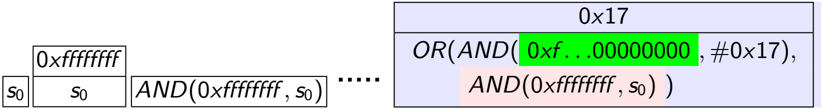


Optimization Example

```
PUSH1 0x17, DUP1, SLOAD, PUSH4 0xffffffff, NOT, AND, PUSH4
0xffffffff, SWAP3, SWAP1, SWAP3, AND, SWAP2, SWAP1, SWAP2,OR, SWAP1
```

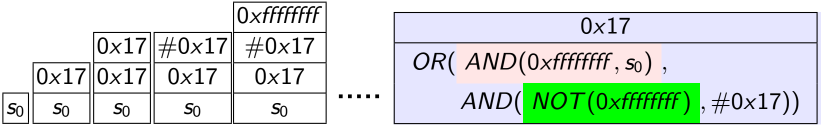



```
PUSH4 0xffffffff, AND PUSH32, 0xff...00000000, PUSH1 0x17, SLOAD,
AND, OR, PUSH1 0x17
```

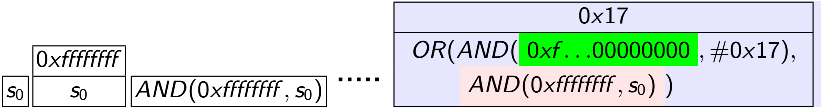


Optimization Example

```
PUSH1 0x17, DUP1, SLOAD, PUSH4 0xffffffff, NOT, AND, PUSH4
0xffffffff, SWAP3, SWAP1, SWAP3, AND, SWAP2, SWAP1, SWAP2,OR, SWAP1
```



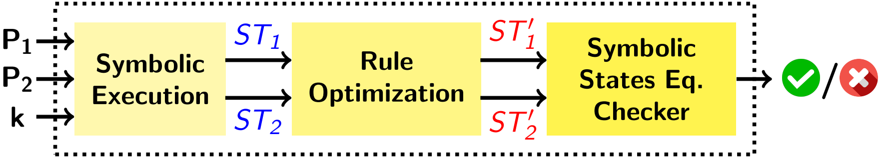

```
PUSH4 0xffffffff, AND PUSH32, 0xff...00000000, PUSH1 0x17, SLOAD,
AND, OR, PUSH1 0x17
```



The final stacks are equivalent due to:

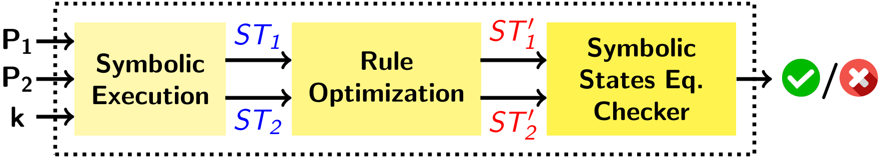
- Simplification rule: $NOT(0xffffffff) = 0xf\ldots00000000$.
- Comutativity of OR

Verification of Results




```
Theorem opt_correct:
  forall (p1 p2 : prog) (k: nat),
    eq_chkr p1 p2 k = true →
      forall (s : stack), length s = k → evmSem p1 s = evmSem p2 s.
```

Verification of Results



ST_1:

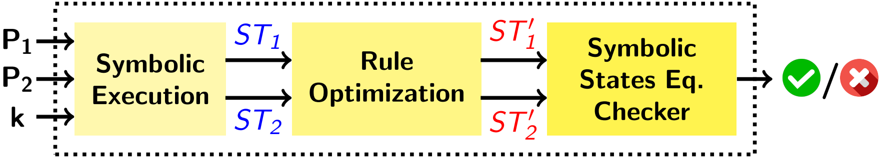
0x17
$OR(AND(0xffffffff, s_0), AND(NOT(0xffffffff), \#0x17))$

ST_2:

0x17
$OR(AND(0xf\ldots00000000, \#0x17), AND(0xffffffff, s_0))$

```
Theorem opt_correct:
  forall (p1 p2 : prog) (k: nat),
    eq_chkr p1 p2 k = true →
      forall (s : stack), length s = k → evmSem p1 s = evmSem p2 s.
```

Verification of Results



ST_1:

0x17
$OR(AND(0xffffffff, s_0), AND(NOT(0xffffffff), \#0x17))$

ST'_1:

0x17
$OR(AND(0xffffffff, s_0), AND(0xf\ldots00000000, \#0x17))$

ST_2:

0x17
$OR(AND(0xf\ldots00000000, \#0x17), AND(0xffffffff, s_0))$

ST'_2:

0x17
$OR(AND(0xf\ldots00000000, \#0x17), AND(0xffffffff, s_0))$

```
Theorem opt_correct:
  forall (p1 p2 : prog) (k: nat),
    eq_chkr p1 p2 k = true →
      forall (s : stack), length s = k → evmSem p1 s = evmSem p2 s.
```

Conclusions & Future Work

Conclusions

- We have introduced a framework for the optimization of smart contracts based on formal methods and made this approach feasible for practical use

Conclusions

- We have introduced a framework for the optimization of smart contracts based on formal methods and made this approach feasible for practical use
- Superoptimization has very interesting applications:
  - Extra Layer of Optimization
  - Learning New Peephole Optimizations
  - Development of New Optimization Techniques

Conclusions

- We have introduced a framework for the optimization of smart contracts based on formal methods and made this approach feasible for practical use
- Superoptimization has very interesting applications:
  - Extra Layer of Optimization
  - Learning New Peephole Optimizations
  - Development of New Optimization Techniques
- It has been used to improve the **solc** compiler optimization algorithm!

Conclusions

- We have introduced a framework for the optimization of smart contracts based on formal methods and made this approach feasible for practical use
- Superoptimization has very interesting applications:
  - Extra Layer of Optimization
  - Learning New Peephole Optimizations
  - Development of New Optimization Techniques
- It has been used to improve the **solc** compiler optimization algorithm!
- Four grants given by the Ethereum Foundation:

Conclusions

- We have introduced a framework for the optimization of smart contracts based on formal methods and made this approach feasible for practical use
- Superoptimization has very interesting applications:
  - Extra Layer of Optimization
  - Learning New Peephole Optimizations
  - Development of New Optimization Techniques
- It has been used to improve the **solc** compiler optimization algorithm!
- Four grants given by the Ethereum Foundation:
  - Adaptation of GASOL into the **solc** compiler output and memory extension enhancement (Finished!)

Conclusions

- We have introduced a framework for the optimization of smart contracts based on formal methods and made this approach feasible for practical use
- Superoptimization has very interesting applications:
  - Extra Layer of Optimization
  - Learning New Peephole Optimizations
  - Development of New Optimization Techniques
- It has been used to improve the **solc** compiler optimization algorithm!
- Four grants given by the Ethereum Foundation:
  - Adaptation of GASOL into the **solc** compiler output and memory extension enhancement (Finished!)
  - Formal verification of optimization results (Foryu project, ongoing)

Conclusions

- We have introduced a framework for the optimization of smart contracts based on formal methods and made this approach feasible for practical use
- Superoptimization has very interesting applications:
  - Extra Layer of Optimization
  - Learning New Peephole Optimizations
  - Development of New Optimization Techniques
- It has been used to improve the **solc** compiler optimization algorithm!
- Four grants given by the Ethereum Foundation:
  - Adaptation of GASOL into the **solc** compiler output and memory extension enhancement (Finished!)
  - Formal verification of optimization results (Foryu project, ongoing)
  - Integration by means of a greedy algorithm into the Yul to EVM compilation pipeline (Grey project, ongoing)

Future Work

- Adapt the framework to synthesize EVM bytecode from Yul code

Future Work

- Adapt the framework to synthesize EVM bytecode from Yul code
- Incorporate more precise information based on Data-Flow analysis

Future Work

- Adapt the framework to synthesize EVM bytecode from Yul code
- Incorporate more precise information based on Data-Flow analysis
- Study more applications of AI within superoptimization

Future Work

- Adapt the framework to synthesize EVM bytecode from Yul code
- Incorporate more precise information based on Data-Flow analysis
- Study more applications of AI within superoptimization
- Enable Associative-Commutative Reasoning in a SMT Solver to produce an alternative encoding

Future Work

- Adapt the framework to synthesize EVM bytecode from Yul code
- Incorporate more precise information based on Data-Flow analysis
- Study more applications of AI within superoptimization
- Enable Associative-Commutative Reasoning in a SMT Solver to produce an alternative encoding
- Superoptimization of other Stack-Based Languages

Thank you for your attention!