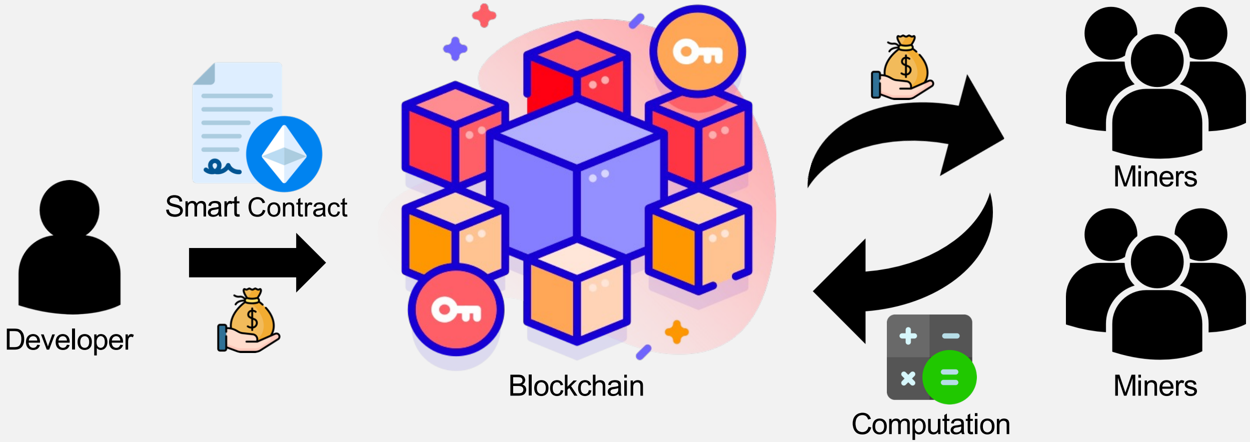
Synthesis-Powered Optimization of Smart Contracts via Data Type Refactoring

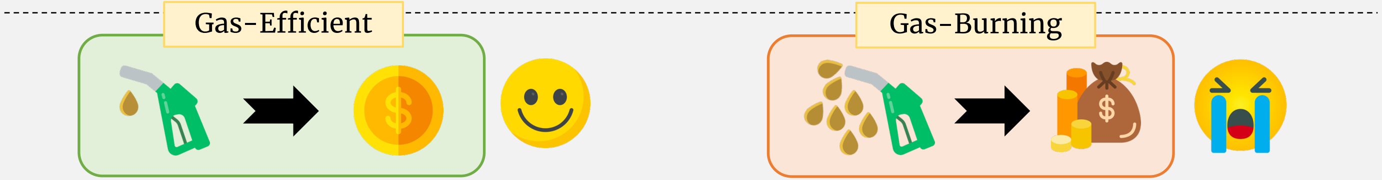
Yanju Chen*[15], Yuepeng Wang*[2], Maruth Goyal[3], James Dong[4], Yu Feng[15], Isil Dillig[35]

UCSB SFU SIMON FRASER UNIVERSITY Stanford University TEXAS The University of Texas at Austin Veridise.

*equal contribution
1. University of California, Santa Barbara 2. Simon Fraser University 3. Stanford University 4. The University of Texas at Austin 5. Veridise Inc.

Gas Optimization of Smart Contracts





Developers typically invest significant effort in optimizing their code and making it as gas-efficient as possible.

Related Approaches

- Bytecode Superoptimization
  - SYRUP[1], GASOL[2]
- Anti-Pattern Detection
  - GASPER[3], GasReducer[4]

Reducing gas usage of some contracts requires significant changes to data layout, which is not addressed by any prior work.

[1] Synthesis of Super-Optimized Smart Contracts Using Max-SMT. *Elvira Albert, Pablo Gordillo, Albert Rubio, Maria A. Schett*. In CAV'20.
[2] GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. *Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, Albert Rubio*. In TACAS'20.
[3] Under-optimized smart contracts devour your money. *Ting Chen, Xiaoqi Li, Xiapu Luo, Xiaosong Zhang*. In SANER'17.
[4] Towards saving money in using smart contracts. *Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, Xiaosong Zhang*. In ICSE-NIER'18.

Example

```
contract CreditDAO {
    struct Election {
        address maxVotes;
        uint nextCandidateIndex;
        mapping(address => bool) candidates;
        mapping(address => bool) userHasVoted;
        mapping(uint => uint) candidateVotes;
        uint numMaxVotes;
        uint idProcessed;
    }
    uint public nextEId;
    mapping(uint => Election) public elections;
    constructor() public {
        nextEId++;
    }
    function sumbitForElection() public {
        elections[nextEId-1].nextCandidateIndex++;
        elections[nextEId-1].candidates[msg.sender] = true;
    }
    function vote(uint candidateId) public {
        elections[nextEId-1].candidateVotes[candidateId] += 1;
        elections[nextEId-1].userHasVoted[msg.sender] = true;
    }
    function finishElections(uint _iterations) public {
        uint currentVotes;
        Election election = elections[nextEId-1];
        uint nextId = election.idProcessed;
        for (uint cnt = 0; cnt < _iterations; cnt++) {
            currentVotes = election.candidateVotes[nextId];
            if (currentVotes > election.numMaxVotes) {
                election.numMaxVotes = currentVotes;
            }
            nextId++;
        }
        election.idProcessed = nextId;
    }
}
```

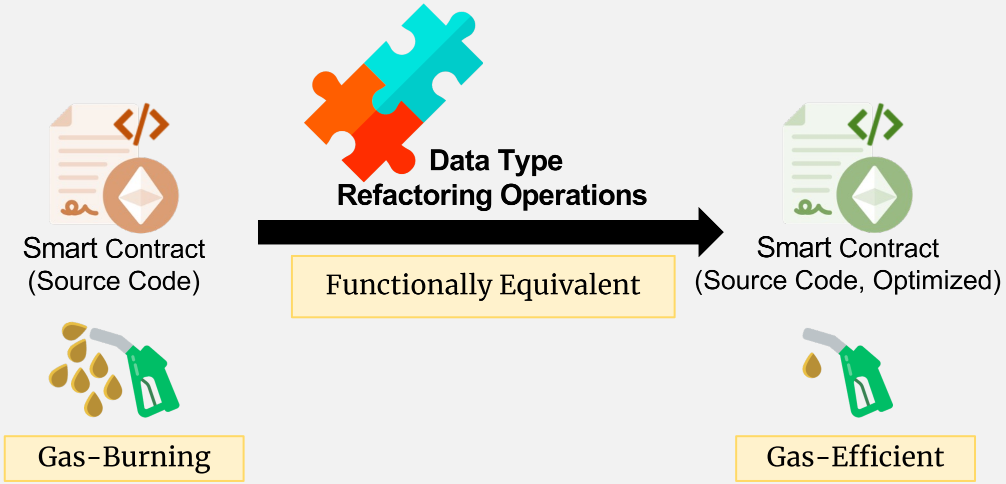
Figure. A motivating real-world smart contract that is not gas efficient.

Example

```
contract CreditDAO {
    struct Election {
        address maxVotes;
        uint nextCandidateIndex;
        mapping(address => bool) candidates;
        mapping(address => bool) userHasVoted;
        mapping(uint => uint) candidateVotes;
        uint numMaxVotes;
        uint idProcessed;
    }
    uint public nextEId;
    mapping(uint => Election) public elections;
    constructor() public {
        nextEId++;
    }
    function sumbitForElection() public {
        elections[nextEId-1].nextCandidateIndex++;
        elections[nextEId-1].candidates[msg.sender] = true;
    }
    function vote(uint candidateId) public {
        elections[nextEId-1].candidateVotes[candidateId] += 1;
        elections[nextEId-1].userHasVoted[msg.sender] = true;
    }
    function finishElections(uint _iterations) public {
        uint currentVotes;
        Election election = elections[nextEId-1];
        uint nextId = election.idProcessed;
        for (uint cnt = 0; cnt < _iterations; cnt++) {
            currentVotes = election.candidateVotes[nextId];
            if (currentVotes > election.numMaxVotes) {
                election.numMaxVotes = currentVotes;
            }
            nextId++;
        }
        election.idProcessed = nextId;
    }
}
```

Figure. A motivating real-world smart contract that is not gas efficient.

Gas Optimization via Type Refactoring



Example

How about a syntax-based rewriting?

```
contract CreditDAO {
    struct Election {
        address maxVotes;
        uint nextCandidateIndex;
        mapping(address => bool) candidates;
        mapping(address => bool) userHasVoted;
        mapping(uint => uint) candidateVotes;
        uint numMaxVotes;
        uint idProcessed;
    }
```

```
contract CreditDAO {
    struct Election {
        address maxVotes;
        uint nextCandidateIndex;
        mapping(address => Participant) userMap;
        mapping(uint => uint) candidateVotes;
    }
    struct Count {
        uint numMaxVotes;
        uint idProcessed;
    }
    struct Participant {
        bool isCandidate;
        bool hasVoted;
    }
```

1. It's often difficult to determine which of the rewriting strategies would result in equivalent code;
2. It can't ensure gas-optimality of the generated code.

Example

```
contract CreditDAO {
    struct Election {
        address maxVotes;
        uint nextCandidateIndex;
        mapping(address => bool) candidates;
        mapping(address => bool) userHasVoted;
        mapping(uint => uint) candidateVotes;
        uint numMaxVotes;
        uint idProcessed;
    }
    uint public nextEId;
    mapping(uint => Election) public elections;
    constructor() public {
        nextEId++;
    }
    function sumbitForElection() public {
        elections[nextEId-1].nextCandidateIndex++;
        elections[nextEId-1].candidates[msg.sender] = true;
    }
    function vote(uint candidateId) public {
        elections[nextEId-1].candidateVotes[candidateId] += 1;
        elections[nextEId-1].userHasVoted[msg.sender] = true;
    }
    function finishElections(uint _iterations) public {
        uint currentVotes;
        Election election = elections[nextEId-1];
        uint nextId = election.idProcessed;
        for (uint cnt = 0; cnt < _iterations; cnt++) {
            currentVotes = election.candidateVotes[nextId];
            if (currentVotes > election.numMaxVotes) {
                election.numMaxVotes = currentVotes;
            }
            nextId++;
        }
        election.idProcessed = nextId;
    }
}
```

```
contract CreditDAO {
    struct Election {
        address maxVotes;
        uint nextCandidateIndex;
        mapping(address => Participant) userMap;
        mapping(uint => uint) candidateVotes;
    }
    struct Count {
        uint numMaxVotes;
        uint idProcessed;
    }
    struct Participant {
        bool isCandidate;
        bool hasVoted;
    }
    uint public nextEId;
    mapping(uint => Election) public elections;
    mapping(uint => Count) public counts;
    constructor() public {
        nextEId++;
    }
    function sumbitForElection() public {
        elections[nextEId-1].nextCandidateIndex++;
        elections[nextEId-1].userMap[msg.sender].isCandidate = true;
    }
    function vote(uint candidateId) public {
        elections[nextEId-1].candidateVotes[candidateId] += 1;
        elections[nextEId-1].userMap[msg.sender].hasVoted = true;
    }
    function finishElections(uint _iterations) public {
        uint currentVotes;
        Election election = elections[nextEId-1];
        Count count = counts[nextEId-1];
        uint nextId = count.idProcessed;
        for (uint cnt = 0; cnt < _iterations; cnt++) {
            currentVotes = election.candidateVotes[nextId];
            if (currentVotes > count.numMaxVotes) {
                count.numMaxVotes = currentVotes;
            }
            nextId++;
        }
        count.idProcessed = nextId;
    }
}
```

Figure. Differences between the smart contract ***before (left)*** and ***after (right)*** refactoring for gas optimization (~30% reduction).

Reducing the gas usage requires significant changes to data layouts ***and*** re-implementing significant part of the contract code.

Overview of Solidare

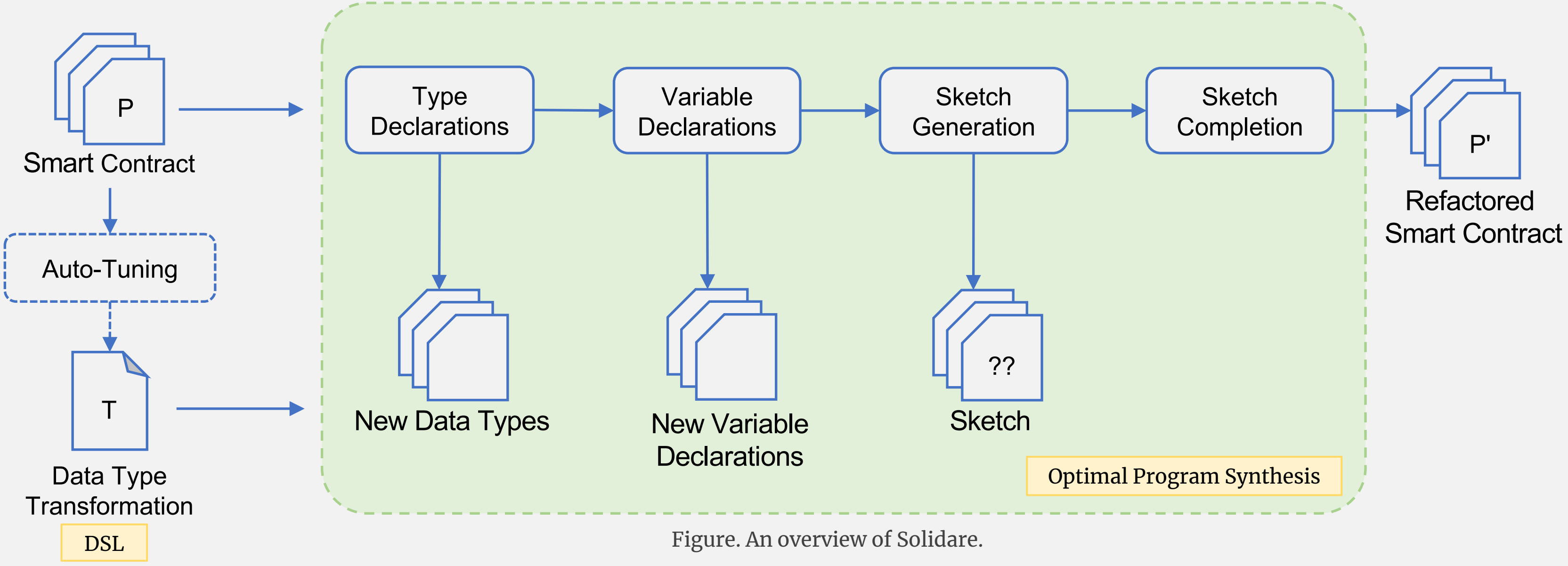


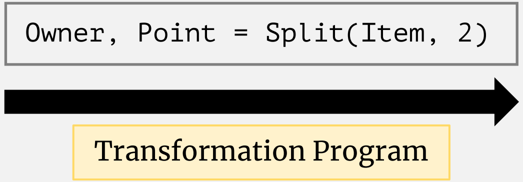
Figure. An overview of Solidare.

A Running Example

```
contract SimplePoint {
    struct Item {
        bool activated;
        address owner;
        uintX x;
        uintY y;
    }
    mapping(uint => Item) items;
    function set(uint i, uintX _x, uintY _y) public {
        items[i].x = _x;
        items[i].y = _y;
    }
    function getX(uint i) public view returns (uintX) {
        return items[i].x;
    }
    function getY(uint i) public view returns (uintY) {
        return items[i].y;
    }
    // more functions omitted
    // ...
}
```

Step 1. Type Declarations

```
struct Item {
    bool activated;
    address owner;
    uintX x;
    uintY y;
}
```




```
struct Point {
    uintX x;
    uintY y;
}
struct Owner {
    bool activated;
    address addr;
}
```

$$
\begin{array}{rcl}
\text{Trans. } \mathcal{T} & ::= & s \mid \mathcal{T};\mathcal{T} \\
\text{Stmt. } s & ::= & S \leftarrow \textbf{Wrap}(\tau, \ldots, \tau) \mid \textbf{Unwrap}(S) \mid (S, S) \leftarrow \textbf{Split}(S, c) \\
& \mid & S \leftarrow \textbf{Merge}(S, S) \mid S \leftarrow \textbf{Reorder}(S, c, c)
\end{array}
$$

$$c \in \textbf{Constant} \quad S \in \textbf{StructName} \quad \tau \in \textbf{Type}$$

Figure. Syntax of the transformation language; see paper for semantics

Step 2. Variable Declarations

```
struct Item {
    bool activated;
    address owner;
    uintX x;
    uintY y;
}
mapping(uint => Item) items;
```

```
Owner, Point = Split(Item, 2)
```

Transformation Program

```
struct Point {
    uintX x;
    uintY y;
}
struct Owner {
    bool activated;
    address addr;
}
mapping(uint => Point) points;
mapping(uint => Owner) owners;
```

Step 3. Code Generation

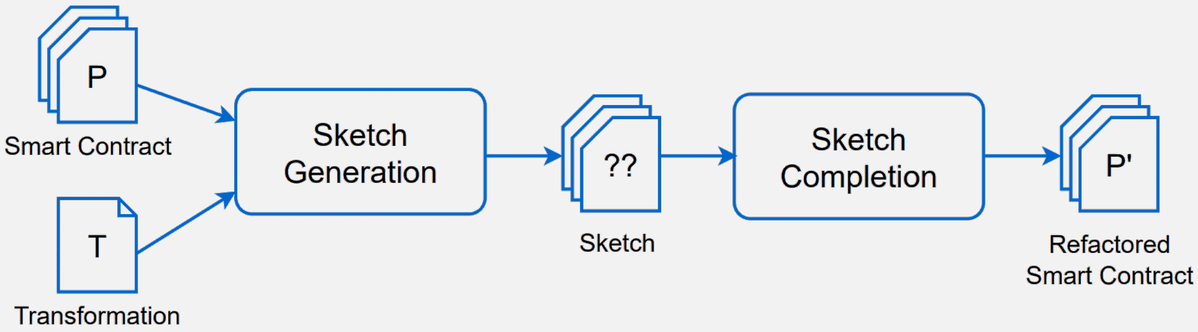


Figure. Workflow of the code generation procedure.

Step 3a. Sketch Generation (Expr.)

```
function set(uint i, uintX _x, uintY _y) public {
    items[i].x = _x;
    items[i].y = _y;
}
```

Stale Expressions

```
Owner, Point = Split(Item, 2)
```

Transformation Program $\mathcal{T} \vdash \Gamma \hookrightarrow \Gamma'$

```
function set(uint i, uintX _x, uintY _y) public {
    if (??1) ??2 = ??3;
    if (??4) ??5 = ??6;
}
```

Each stale expression is replaced with a *hole* whose *domain* includes well-typed expressions.

*Please see the paper for detailed sketch generation rules.

Type Environment Γ

$\Gamma \vdash$ items[i].x : Item.uintX
$\Gamma \vdash$ items[i].y : Item.uintY
$\Gamma \vdash$ _x: Item.uintX
$\Gamma \vdash$ _y: Item.uintY
…

Type Environment Γ'

$\Gamma \vdash$ points[i].x : Point.uintX
$\Gamma \vdash$ points[i].y : Point.uintY
$\Gamma \vdash$ _x: Point.uintX
$\Gamma \vdash$ _y: Point.uintY
…

Step 3a. Sketch Generation (Stmt.)

```
function set(uint i, uintX _x, uintY _y) public {
    items[i].x = _x;
    items[i].y = _y;
}
```

```
Owner, Point = Split(Item, 2)
```

Transformation Program $\mathcal{T} \vdash \Gamma \hookrightarrow \Gamma'$

```
function set(uint i, uintX _x, uintY _y) public {
    if (??1) ??2 = ??3;
    if (??4) ??5 = ??6;
}
```

Replace each statement *s* with a stale expression with a conditional statement: **if (??{⊤, ⊥}) then s' else skip**.

*Please see the paper for detailed sketch generation rules.

```
m[0] = Point(x,y);
```

```
Unwrap(Point)
```

```
if (??1) ??2 = ??3; // x
if (??4) ??5 = ??6; // y
```

Some statements become redundant after transformation; removing them saves gas.

Step 3b. Sketch Completion (Alg. & Enc.)

- Max-SAT Encoding
  - Hard Constraints
    - Every hole should be instantiated with ***exactly one*** expression in its domain.
    - Different occurrences of ***same*** source expression are transformed into the ***same*** target expression.
  - Soft Constraints (Proxy Metric of Gas Usage)
    - Minimizing blockchain variables
    - Minimizing statements

Algorithm 1 Sketch Completion

1: **procedure** CompleteSketch($\mathcal{S}, \mathcal{P}$)
Input: Sketch $\mathcal{S}$, Source program $\mathcal{P}$
Output: Target program $\mathcal{P}'$ or $\bot$ to indicate failure
2: $\Phi \leftarrow$ Encode($\mathcal{S}$);
3: **while** SAT(Φ) **do**
4: $\mathcal{M} \leftarrow$ GetModel(Φ);
5: $\mathcal{P}' \leftarrow$ Instantiate($\mathcal{S}, \mathcal{M}$);
6: **if** $\mathcal{P}' \simeq \mathcal{P}$ **then return** $\mathcal{P}'$;
7: $\Phi \leftarrow \Phi \wedge$ Block($\mathcal{P}, \mathcal{S}, \mathcal{M}$);
8: **return** $\bot$;

*Please see the paper for detailed encoding and algorithm.

Step 3b. Sketch Completion (MFS)

- Minimal Failing Sub-Contract

Original Contract P

```
contract SimplePoint {
  uint public x = 0; uint public y = 0;
  function set(uint _x, uint _y) public { x = _x; y = _y; }
  function getX() public returns (uint) { return x; }
  function getY() public returns (uint) { return y; } }
```

Transformed Contract (Incorrect)

Minimal Failing Sub-Contract P^*

```
contract SimplePoint {
  uint public x = 0; uint public y = 0;
  function set(uint _x, uint _y) public { x = _x; y = _y; }
  function getX() public returns (uint) { return y; } }
  function getY() public returns (uint) { return y; } }
```

1. P^* only contains a subset of functions in P
2. P^* is not equivalent to P with respect to functions it implements
3. P^* is minimal – removing any functions would make P^* and P equivalent with respect to functions P^* implemented

*Please see the paper for more details.

The key idea is to generalize model $\mathcal{M}$ and add a blocking clause that prevents *many* incorrect programs at the same time.

Evaluation

- SOLIDARE is implemented in a combination of Java and Kotlin, with Sat4J[1] as backend.
- Benchmarks
  - **Etherscan**: 20
    - Contains rich data structures, complicated control flows
    - Wide coverage: auctions, crowd sourcing, decentralized autonomous organizations (DAOs), etc.
  - **GasStation**: 10
    - Most frequently used smart contracts / gas burners
- Experimental Settings
  - Two usage modes: manual + auto-tuner transformations
  - Intel® Xeon® E5-2640@2.60GHz CPU, 128GB Physical Memory
  - Ubuntu 18.04@Docker
  - For more implementation details, please refer to the paper

[1] The sat4j library, release 2.2, system description. *Daniel Le Berre, Anne Parrain*. In Journal on Satisfiability, Boolean Modeling and Computation 7. 2010

Evaluation

RQ1: Is SOLIDARE able to generate equivalent code for different data layouts?

Yes.

Averaged running time: 21.1s
Medium running time: 0.9s

Major time cost: sketch completion (including equivalence checking)

	ID	Contract	LOC	# Funcs	# Trans	Sketch Time (s)	Completion Time (s)	Max Diff	Avg Diff
Etherscan	1	Announcement	112	7	2	0.2	0.2	23	17.5
	2	Auction	964	70	1	3.7	7.7	34	34.0
	3	BdpImageStorage	258	27	2	0.1	0.4	32	32.0
	4	BinaryOption	916	20	1	0.4	1.4	31	31.0
	5	Congress	163	9	3	1.2	1.6	66	34.7
	6	CreditDAO	111	14	2	0.4	0.4	54	50.0
	7	CryptoTask	255	17	3	0.3	0.3	12	8.3
	8	DAOG2X	319	19	3	0.7	1.9	24	23.0
	9	EMPresale	306	30	3	0.7	551.1	57	38.0
	10	EthLottery	132	6	2	0.3	0.2	22	21.5
	11	EtherRacing	250	20	2	1.2	6.2	32	32.0
	12	FTICrowdsale	553	17	1	0.1	0.3	9	9.0
	13	JanKenPon	510	40	1	17.0	2.7	47	47.0
	14	Kingdom	189	13	3	0.6	3.5	64	44.7
	15	Oryza	152	7	2	1.0	1.4	21	20.0
	16	PollManager	473	12	2	3.0	3.1	17	17.0
	17	Slaughter3D	287	26	1	0.7	2.2	22	22.0
	18	SplitStealContract	465	28	2	5.0	3.9	24	23.5
	19	TwoXJackpot	222	15	1	0.4	1.3	14	14.0
	20	moduleToken	392	21	2	0.6	0.8	18	18.0
GasStation	21	MetaMasks	597	88	1	0.1	0.2	18	18.0
	22	MoonStaking	525	65	1	0.1	0.4	19	19.0
	23	MoonStakingForTax	842	120	1	0.1	0.3	24	24.0
	24	MASTERPLAN	494	57	1	0.1	0.4	21	21.0
	25	MasterInu	758	149	1	0.1	0.9	15	15.0
	26	MetaPunkController2022	1586	446	1	0.2	0.2	14	14.0
	27	KaijuFrenz	924	99	1	0.1	0.2	25	25.0
	28	EMOBUDDIES	852	101	1	0.1	0.2	15	15.0
	29	GemSwap	528	76	1	0.1	0.3	16	16.0
	30	LL420Reveal	131	16	1	0.1	0.1	10	10.0

Table. Statistics about benchmarks and results of running time.

Evaluation

RQ2: Can we reduce the gas usage of real-world smart contracts through data type refactoring?

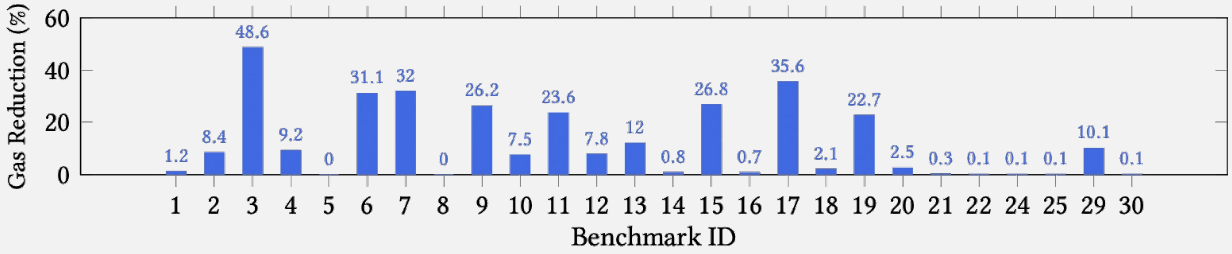


Figure. Gas reduction in benchmarks.

Yes.

Etherscan Dataset: 18/20 have improvement, avg. gas saving is 16%.

GasStation Dataset: 6/10 have improvement, gas saving is 0.1% ~ 10%.

Most benchmarks in GasStation are digital tokens, which require more complex program logic and less complicated data layout.

Evaluation

RQ3: How much manual effort does SOLIDARE save developers?

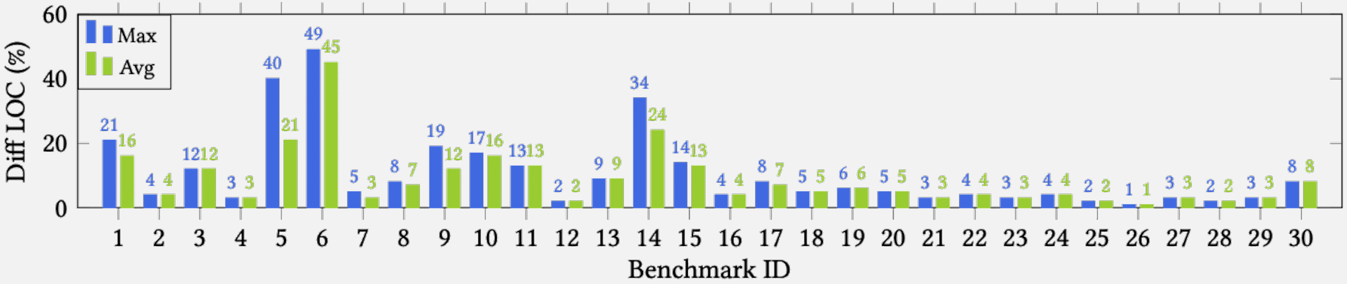


Figure. Diff size as percentage of the lines of code in original contracts. **Max**: Statistics of transformation the requires the most changes; **Avg**: Averaged diff ratio per benchmark across all transformations.

On average, 25% of the lines of code (~avg. 53 lines) need to be modified.

The largest diff size could be 49% and 40%.

Evaluation

RQ4: How does our sketch completion method compare with simpler baselines?

Timeout: 20min

Ablative Variants:

- SOLIDARE solves **100%**
- SOLIDARE-NOMFS solves 22% less
- SOLIDARE-NOSOFT solves 10% less
- SOLIDARE-BASELINE solves 24% less

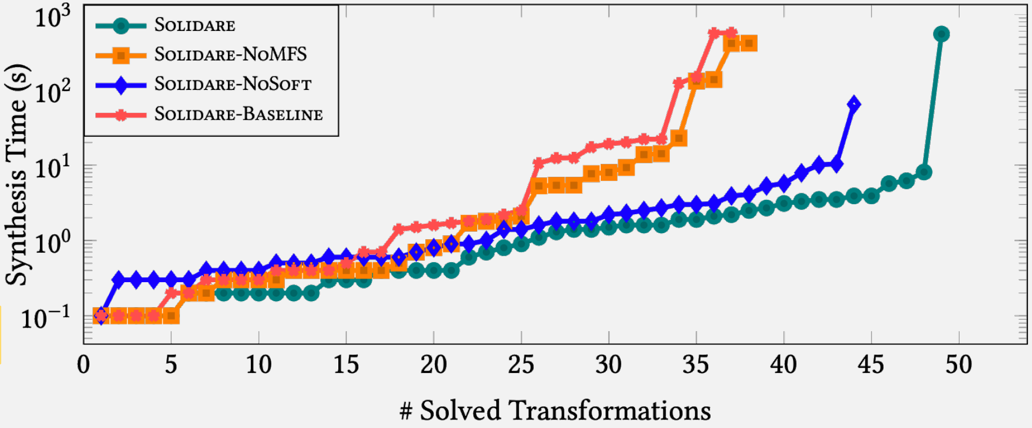


Figure. Comparing SOLIDARE against baselines. y-axis is on log-scale.

Our MaxSAT-based sketch solver that utilizes minimal failing sub-contracts significantly outperforms other baselines.

Evaluation

RQ5: Is SOLIDARE's auto-tuner able to automatically find gas-saving refactorings?

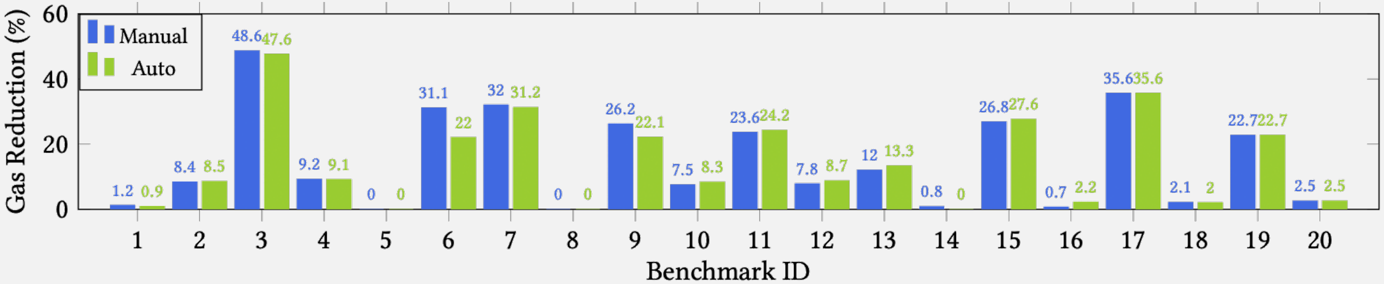


Figure. Comparison of gas reduction between manual refactoring and autotuning.

Yes.

The auto-tuner can automatically reduce gas usage for 17/20 (85%) benchmarks.

For 13/20 (65%), the reduction is >5%.

For 9/20 (45%), the reduction is >10%.

For 7/20 (65%), auto-tuner is better than manual refactoring.

Evaluation

RQ6: How does SOLIDARE compare with other gas optimization tools?

- SYRUP[1]: Bytecode Superoptimization

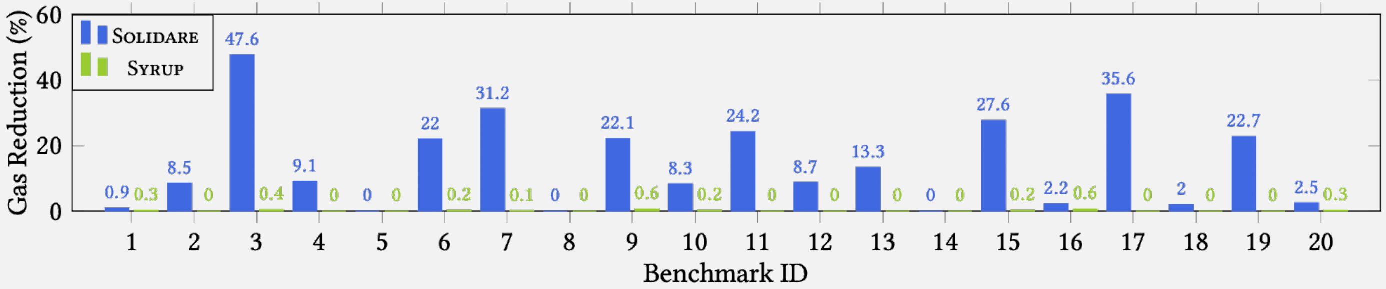


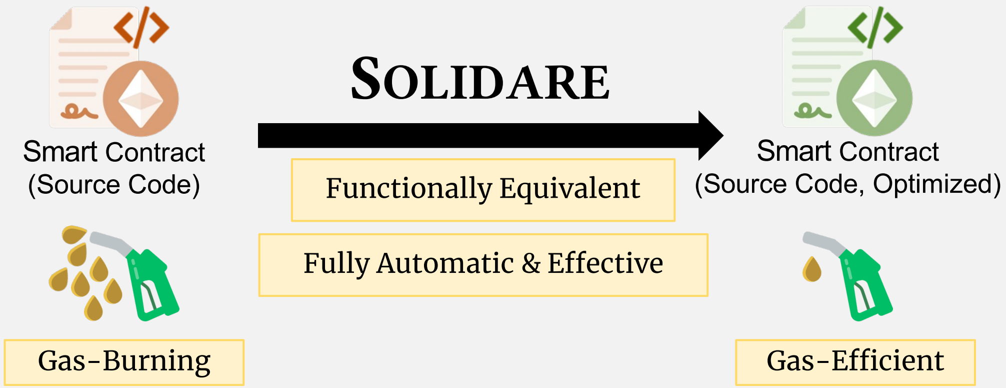
Figure. Comparing SOLIDARE and SYRUP.

SYRUP focuses on optimizing arithmetic operations within a basic block, but more significant gas savings require changing in data layout.

Nonetheless, we believe the types of optimizations performed by SYRUP are complementary SOLIDARE's.

[1] Synthesis of Super-Optimized Smart Contracts Using Max-SMT. *Elvira Albert, Pablo Gordillo, Albert Rubio, Maria A. Schett.* In CAV'20.

Conclusion


Smart Contract
(Source Code)
Gas-Burning
SOLIDARE
Functionally Equivalent
Fully Automatic & Effective
Smart Contract
(Source Code, Optimized)
Gas-Efficient