Supplemental Material for the FSE 2026 paper: "SmarTrim: Symbolic Execution for Smart Contracts Powered by Redundant Transaction-Sequence Pruning"

HYEGEUN SONG, Korea University, Republic of Korea
JISEONG HAN, Korea University, Republic of Korea
SUNBEOM SO*, Korea University, Republic of Korea

A Examples with Composite Data Types (§ 2.2)

While Section 2.2 demonstrated our approach on a simple contract with scalar variables, it also applies to contracts with composite data types such as mappings, arrays, and structures. We present examples of subsumption conditions generated by SmarTrim for these data types.

Mappings. Consider the following contract:

```
1  contract ExampleMapping {
2    mapping(uint=>uint) m;
3
4    function set1 (uint k, uint z) { m[k] = z; }
5    function set2 (uint i, uint j, uint x, uint y) { m[i] = x; m[j] = y; }
6  }
```

It shows an example of contracts with Solidity mappings, in which a transaction sequence $p : t_0 \cdot t_1$ is subsumed by $q : t_0 \cdot t_2$ where

$$t_0\text{: constructor( )},\ t_1\text{: set1(k, z)},\ t_2\text{: set2(i, j, x, y)}$$

We omit `msg.sender` for simplicity. As we model Solidity mappings with the theory of arrays, when we try to check subsumed (p, q), we check the validity of the following subsumption condition:

$$\exists k, z, m'.\ \underbrace{m' = \mathsf{K}_0 \wedge m = m'\langle k \triangleleft z\rangle}_{\phi_p} \rightarrow \exists i, j, x, y, m', m''.\ \underbrace{m' = \mathsf{K}_0 \wedge m'' = m'\langle i \triangleleft x\rangle \wedge m = \langle m'' \triangleleft y\rangle}_{\phi_q}$$

where K_0 is a constant array whose elements are all initialized to 0, and ϕ_p (resp. ϕ_q) is the state condition of p (resp. q). We omit entry variables (Section 3.1) for brevity.

Arrays. Consider the contract below:

```
1  contract ExampleArray {
2    uint[] a;
3
4    function push10 ( ) { a.push(10); }
5    function pushX (uint x) { a.push(x); }
6  }
```

*Corresponding author

Authors' Contact Information: Hyegeun Song, Korea University, Seoul, Republic of Korea, hyegeun_song@korea.ac.kr; Jiseong Han, Korea University, Seoul, Republic of Korea, jiseong104@korea.ac.kr; Sunbeom So, Korea University, Seoul, Republic of Korea, sunbeom_so@korea.ac.kr.

This contract represents an example of contracts with arrays, in which a transaction sequence $p : t_0 \cdot t_1$ is subsumed by $q : t_0 \cdot t_2$ where

$$t_0\text{: } \texttt{constructor( )},\ t_1\text{: } \texttt{push10( )},\ t_2\text{: } \texttt{pushX(x)}$$

We model Solidity arrays (as well as dynamic arrays, which support variable-length storage) using the theory of arrays. Additionally, to model the length of Solidity arrays, we introduce an additional array-typed variable L that takes an array as input and returns its length. We treat L as a global variable. Therefore, to verify subsumed (p, q), we check the validity of the following subsumption condition:

$$\exists a'.\ \underbrace{a' = \mathsf{K}_0 \wedge L[a'] = 0 \wedge a = a'\langle 0 \triangleleft 10\rangle \wedge L[a] = L[a'] + 1}_{\phi_p} \rightarrow$$
$$\exists x, a'.\ \underbrace{a' = \mathsf{K}_0 \wedge L[a'] = 0 \wedge a = a'\langle 0 \triangleleft x\rangle \wedge L[a] = L[a'] + 1}_{\phi_q}$$

Structures. Consider the following contract:

```
1  contract ExampleStruct {
2    struct Point { uint x; uint y; }
3
4    Point r;
5
6    function setPOnALine (uint a) { r.x = a; r.y = a; }
7    function setP (uint b, uint c) { r.x = b; r.y = c; }
8  }
```

The contract describes an example of contracts with structures, in which a transaction sequence $p : t_0 \cdot t_1$ is subsumed by $q : t_0 \cdot t_2$ where

$$t_0\text{: } \texttt{constructor( )},\ t_1\text{: } \texttt{setPOnALine(a)},\ t_2\text{: } \texttt{setP(b, c)}$$

We model Solidity structures as arrays; that is, if an expression `s.member` exists, we interpret it as $member[s]$ where $member$ is an array-typed global variable and s is an integer-typed structure identifier. Since different structure objects must have distinct identifiers, we assign identifiers to each structure object during the preprocessing phase. Since there is only one structure object, r, in `ExampleStruct`, we assign r's identifier as 0. Therefore, when verifying subsumed (p, q), we check the validity of the following subsumption condition:

$$\exists a, x', y'.\ \underbrace{r = 0 \wedge x' = \mathsf{K}_0 \wedge y' = \mathsf{K}_0 \wedge x = x'\langle r \triangleleft a\rangle \wedge y = y'\langle r \triangleleft a\rangle}_{\phi_p} \rightarrow$$
$$\exists b, c, x', y'.\ \underbrace{r = 0 \wedge x' = \mathsf{K}_0 \wedge y' = \mathsf{K}_0 \wedge x = x'\langle r \triangleleft b\rangle \wedge y = y'\langle r \triangleleft c\rangle}_{\phi_q}$$

B Soundness Proof of NoDef (§ 4.2.1)

We prove the soundness of the validity template below:

$$\frac{\mathrm{Def}\,(t) = \emptyset}{\mathrm{subsumed}\,(s \cdot t, s)}\ (\textsc{NoDef})$$

Proof. Rewriting the goal and a proof by contradiction. By Definition 3.2, our goal is to prove the validity of the formula

$$\phi : \exists \mathbf{X}.P \rightarrow \exists \mathbf{Y}.Q \tag{1}$$

where $Q = \mathsf{ST}(s)$, $P = \mathsf{ST}(s \cdot t) = \mathsf{ST}'(t)(\mathsf{ST}(s)) = \mathsf{ST}'(t)(Q)$, $\mathbf{X} = \mathsf{FV}(P) \setminus G$, and $\mathbf{Y} = \mathsf{FV}(Q) \setminus G$. The hypothesis $\mathsf{Def}\,(t) = \emptyset$ ensures that the symbolic execution of t just adds a new constraint R without renaming variables in Q. Thus, we have $P = \mathsf{ST}'(t)(Q) = Q \wedge R$. Rewriting (1) reduces our goal to proving the validity of:

$$\phi : \exists \mathbf{X}.Q \wedge R \rightarrow \exists \mathbf{Y}.Q \tag{2}$$

Now suppose we have a falsifying model m of (2):

$$\text{(a) } m \models \exists \mathbf{X}.Q \wedge R, \quad \text{(b) } m \not\models \exists \mathbf{Y}.Q \tag{3}$$

Let us write $m' : m \triangleleft \{x_1 \mapsto v_1, \cdots, x_n \mapsto v_n\}$ to denote a variant [2] of m, where m' agrees with m except possibly at $x_1, \cdots, x_n$. From (a) and the semantics of $\exists$ [2], we have

$$m \triangleleft \{q_1 \mapsto v_1, \cdots, q_i \mapsto v_i, r_1 \mapsto v'_1, \cdots, r_j \mapsto v'_j\} \models Q \wedge R \tag{4}$$

for some $v_1, \cdots, v_i$ and $v'_1, \cdots, v'_j$, where $\{q_1, \cdots, q_i\} = \mathsf{FV}(Q) \setminus G$ and $\{r_1, \cdots, r_j\} = \mathsf{FV}(R) \setminus G$. Since local variables in different transactions are disjoint, we have

$$\{q_1, \cdots, q_i\} \cap \{r_1, \cdots, r_j\} = \emptyset \tag{5}$$

From (4) and (5), we have

$$m \triangleleft \{q_1 \mapsto v_1, \cdots, q_i \mapsto v_i\} \models Q \tag{6}$$

From (b) and the semantics of $\exists$, we have

$$m \triangleleft \{q_1 \mapsto v_1, \cdots, q_i \mapsto v_i\} \not\models Q \tag{7}$$

which is contradictory to (6). Thus we conclude that both (1) and (2) are valid. □

C Soundness Proof of Inter (§ 4.2.1)

We prove the soundness of the validity template Inter:

$$\frac{\begin{array}{c} Y_1 : \mathsf{Def}\,(t_1) \cap (\mathsf{Def}\,(t_2) \cup \mathsf{Use}\,(t_2)) = \emptyset \\ Y_2 : \mathsf{Def}\,(t_2) \cap (\mathsf{Def}\,(t_1) \cup \mathsf{Use}\,(t_1)) = \emptyset \end{array}}{\mathsf{subsumed}\,(s \cdot t_2 \cdot t_1, s \cdot t_1 \cdot t_2)}\ \text{(Inter)}$$

Proof. Proof based on equational rewriting. Following Definition 3.2, we prove Inter by showing the validity of $\phi : \exists \mathbf{X}.P \rightarrow \exists \mathbf{Y}.Q$ where $\mathbf{X} = \mathsf{FV}(P) \setminus G$, $\mathbf{Y} = \mathsf{FV}(Q) \setminus G$,

$$P = \mathsf{ST}'(t_1) \circ \mathsf{ST}'(t_2)(F), \qquad Q = \mathsf{ST}'(t_2) \circ \mathsf{ST}'(t_1)(F) \tag{8}$$

and $F = \mathsf{ST}(s)$. For $i \in \{1, 2\}$, let $\sigma_i : \mathsf{Dom}(\sigma_i) \rightarrow \mathsf{Ran}(\sigma_i)$ be a variable substitution by the symbolic execution of the assignments in t_i. Recall from Section 3.2, $\mathsf{Dom}(\sigma_i)$ is a set of variables to rename and $\mathsf{Ran}(\sigma_i)$ is a set of corresponding primed variables, that is,

$$\text{(a) } \mathsf{Dom}(\sigma_i) = \mathsf{Def}\,(t_i), \quad \text{(b) } \mathsf{Ran}(\sigma_i) = \{x' \mid x \in \mathsf{Def}\,(t_i)\}$$

With σ_1 and σ_2, by the one-step symbolic executions, we rewrite P and Q in (8):

$$P = \mathsf{ST}'(t_1)(F\sigma_2 \wedge H_2) \qquad Q = \mathsf{ST}'(t_2)(F\sigma_1 \wedge H_1) \tag{9}$$

where $F\sigma_1$ (resp., $F\sigma_2$) is the application of σ_1 (resp., σ_2) to F, and H_1 (resp., H_2) represents a new constraint generated by the symbolic execution of t_1 (resp., t_2). By the symbolic executions of the last transactions, we rewrite (9):

$$P = (F\sigma_2)\sigma_1 \wedge H_2\sigma_1 \wedge H_1, \qquad Q = (F\sigma_1)\sigma_2 \wedge H_1\sigma_2 \wedge H_2 \tag{10}$$

In Inter, the hypothesis Y_1 (resp., Y_2) ensures that the symbolic execution of t_1 (resp., t_2) does not rename any variables in t_2 (resp., t_1), that is, $\mathsf{Dom}(\sigma_1) \cap \mathsf{FV}(H_2) = \emptyset$ and $\mathsf{Dom}(\sigma_2) \cap \mathsf{FV}(H_1) = \emptyset$, where $\mathsf{Dom}(\sigma_1) = \mathsf{Def}(t_1)$ and $\mathsf{Dom}(\sigma_2) = \mathsf{Def}(t_2)$. It follows that: $H_1\sigma_2 = H_1$ and $H_2\sigma_1 = H_2$. Thus we rewrite (10):

$$P = (F\sigma_2)\sigma_1 \wedge H_1 \wedge H_2, \qquad Q = (F\sigma_1)\sigma_2 \wedge H_1 \wedge H_2 \tag{11}$$

From Y_1 and Y_2, we have $\mathsf{Def}(t_1) \cap \mathsf{Def}(t_2) = \emptyset$, and hence (i) $\mathsf{Dom}(\sigma_1) \cap \mathsf{Dom}(\sigma_2) = \emptyset$. Since any set of global variables and any set of primed variables are disjoint, we have (ii) $\mathsf{Ran}(\sigma_2) \cap \mathsf{Dom}(\sigma_1) = \emptyset$ and (iii) $\mathsf{Ran}(\sigma_1) \cap \mathsf{Dom}(\sigma_2) = \emptyset$. From (i), (ii), and (iii), we have $(F\sigma_2)\sigma_1 = (F\sigma_1)\sigma_2$ in (11). It follows that P and Q are syntactically identical, and therefore ϕ is valid.

□

D Soundness Proof of Prev (§ 4.2.1)

We prove the soundness of Prev:

$$\frac{\begin{array}{l} Y_1 : \mathsf{subsumed}\ (s \cdot t_1, s) \\ Y_2 : \mathsf{Def}\ (t_1) \cap (\mathsf{Def}\ (t_2) \cup \mathsf{Use}\ (t_2)) = \emptyset \\ Y_3 : \mathsf{Def}\ (t_2) \cap (\mathsf{Def}\ (t_1) \cup \mathsf{Use}\ (t_1)) = \emptyset \end{array}}{\mathsf{subsumed}\ (s \cdot t_2 \cdot t_1, s \cdot t_2)}\ (\textsc{Prev})$$

Proof. By applying Inter to Y_2 and Y_3, we have $P : \mathsf{subsumed}(s \cdot t_2 \cdot t_1, s \cdot t_1 \cdot t_2)$. From Y_1, we have $Q : \mathsf{subsumed}(s \cdot t_1 \cdot t_2, s \cdot t_2)$. From P, Q, and Definition 3.2, we have $A(s \cdot t_2 \cdot t_1) \subseteq A(s \cdot t_1 \cdot t_2)$, $A(s \cdot t_1 \cdot t_2) \subseteq A(s \cdot t_2)$, thus deducing $R : A(s \cdot t_2 \cdot t_1) \subseteq A(s \cdot t_2)$. From R and Definition 3.2, we conclude $\mathsf{subsumed}(s \cdot t_2 \cdot t_1, s \cdot t_2)$. □

E Preprocessing for Benchmark Construction (§ 6.1.1)

We describe the methods for collection and refinement of the new 407 contracts, which were originally published by the authors of SPCon [3] and PrettySmart [5].

Vulnerability Scope Alignment. SPCon and PrettySmart detect permission bugs, and collectively the authors marked 911 contracts as vulnerable. However, this is not ground truth in our context, as the existence of permission bugs does not guarantee the existence of EL/SU bugs.

We manually removed 180 contracts that have no operations related to EL/SU bugs (i.e. operations that handle ether transfer or destroy the contract). In Solidity, the following 8 expressions/statements are related to EL/SU:

- `<address>.send(v)`
- `<address>.transfer(v)`
- `<address>.call.value(v)(..)`
- `<address>.call{value: v}(..)`
- `<contract>.<external-function>.value(v)(..)`
- `<contract>.<external-function>{value: v}(v)(..)`
- `suicide(a)`
- `selfdestruct(a)`

```
1  contract Button {
2    uint64 endTime;
3    address payable lastPresser;
4
5    function press( ) public payable {
6      // block.timestamp: current time
7      require(block.timestamp <= endTime);
8      // need to pay at least 0.01 ether to click
9      require(msg.value >= 0.01 ether);
10     lastPresser = msg.sender;
11     endTime = uint64(block.timestamp + 120 seconds);
12   }
13   function close( ) public {
14     // wait at least 120 seconds after the last click
15     require(block.timestamp > endTime);
16     require(lastPresser == msg.sender);
17     // send all money to the winner and die
18     selfdestruct(msg.sender);
19   }
20 }
```

Fig. 1. `Button` contract (simplified for presentation).

Handling Outdated Contracts. We deleted 31 outdated contracts that were originally compiled with `solc` (the Solidity compiler) 0.3 or less.

Deduplication. Due to the nature of blockchain domains, there are contracts that are copied and continuously deployed on the chain with only minor variations. We deduplicated contracts in our dataset due to two reasons: we did not want any particular kind of contract to dominate the characteristics of our benchmarks, and we wanted to reduce the burden of creating ground truths. For deduplication, we first normalized the source code of contracts: we removed all comments from the source code, replaced all hexadecimal integer literals with `0x0` (as hexadecimal integer literals in Solidity typically represent addresses), and replaced the main contract name with a pre-selected string literal. We then compared the similarity of two contracts using the *gestalt pattern matching* algorithm [4] implemented in Python's `difflib` standard library. We iterated through the contracts and excluded a contract c_1 if there existed an already included contract c_2 such that (1) c_1 and c_2 differed in line count by fewer than 20, (2) c_1 had 98.5% or higher similarity to c_2, and (3) if both c_1 and c_2 were artificially modified contracts, they did not share the same origin.

After all these processes we obtained 407 new contracts as a benchmark. We applied the same deduplication criteria to IO Dataset (resp. RE Dataset), and finalized 274 (resp. 49) contracts for the experiment.

F Example: A False Positive Generated by SmarTrim (§ 7)

We present false positives generated by SmarTrim in our comparative experiments (Section 6.1). Consider the `Button` contract [1] (Figure 1), which implements a game where the last caller of `press` within a specified period can claim the entire pot. The winner invokes the `press` function, waits for a specified time, and then claims all funds. While SmarTrim reported EL and SU vulnerabilities at line 18, this winner-takes-all structure is the intended behavior of this gambling contract rather than a flaw.

Other examples of false positives in SmarTrim include: EL (sending Ethers to arbitrary users who correctly provide unpredictable pseudo-random numbers; taking more Ethers than invested amounts due to token price fluctuations; paying interest to investors), SU (`selfdestruct` being

intentionally callable by any user after a designated time; requiring Ether payments to obtain permission for killing contracts), RE (reentrancy that causes only benign state changes).

References

[1] [n. d.]. The `Button` contract. https://etherscan.io/address/0xf7f6b7164fb3ab456715d2e8b84e8baac8bd09a9. Accessed: April 2026.

[2] Aaron R Bradley and Zohar Manna. 2007. *The calculus of computation: decision procedures with applications to verification.* Springer Science & Business Media.

[3] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. 2022. Finding permission bugs in smart contracts with role mining. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 716–727. doi:10.1145/3533767.3534372

[4] John W Ratcliff, David E Metzener, et al. 1988. Pattern matching: The gestalt approach. *Dr. Dobb's Journal* 13, 7 (1988), 46.

[5] Zhijie Zhong, Zibin Zheng, Hong-Ning Dai, Qing Xue, Junjia Chen, and Yuhong Nan. 2024. PrettySmart: Detecting Permission Re-delegation Vulnerability for Token Behaviors in Smart Contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 168, 12 pages. doi:10.1145/3597503.3639140