An Empirical Study on Implicit Constraints in Smart Contract Static Analysis

Tingting Yin
ytt21@mails.tsinghua.edu.cn
Tsinghua University
China

Chao Zhang
chaoz@tsinghua.edu.cn
Tsinghua University
China

Yuandong Ni
nyd17@mails.tsinghua.edu.cn
Tsinghua University
China

Yixiong Wu
wyx18@mails.tsinghua.edu.cn
Tsinghua University
China

Taiyu Wong
wdy19@mails.tsinghua.edu.cn
Tsinghua University
China

Xiapu Luo
daniel.xiapu.luo@polyu.edu.hk
The Hong Kong Polytechnic University
China

Zheming Li
lizm20@mails.tsinghua.edu.cn
Tsinghua University
China

Yu Guo
yu.guo@secbit.io
SECBIT Labs
China

ABSTRACT
Smart contracts are usually financial-related, which makes them attractive attack targets. Many static analysis tools have been developed to facilitate the contract audit process, but not all of them take account of two special features of smart contracts: (1) The external variables, like time, are constrained by real-world factors; (2) The internal variables persist between executions. Since these features import implicit constraints into contracts, they significantly affect the performance of static tools, such as causing errors in reachability analysis and resulting in false positives. In this paper, we conduct a systematic study on implicit constraints from three aspects. First, we summarize the implicit constraints in smart contracts. Second, we evaluate the impact of such constraints on the state-of-the-art static tools. Third, we propose a lightweight but effective mitigation method named ConSym to deal with such constraints and integrate it into OSIRIS. The evaluation result shows that ConSym can filter out 96% of false positives and reduce false negatives by two-thirds.

KEYWORDS
Smart contract, Static analysis, Implicit constraints, Code audit

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 THE IMPLICIT CONSTRAINTS

Many tools have been developed to facilitate the contract audit via static analysis. But most of them can not meet the needs of industrial development due to the high false-positive (FP) and false-negative (FN) rates. One of the reasons is they neglect two special features of smart contracts: (1) The external variables (e.g., time, assets), participate in contract execution; (2) The values of internal variables are decided by the transaction sequences. Without considering these features, static methods usually assume the contract variables can take arbitrary values. However, these features bring three types of implicit constraints on the value range of contract variables.

1 $\delta_1$: Implicit Constraints on External Variables. Smart contracts take inputs from external sources, i.e., transaction properties and blockchain states, which have real-world meanings. For instance, the assets to transfer in transactions (returned by CALLVALUE instruction) cannot exceed the total issued ETH in Ethereum, and the block height (returned by NUMBER instruction) is related to the alive time of the blockchain, which can not be very large.

2 $\delta_2$: Implicit Constraints on Individual Internal Variables. Smart contracts are invoked via transactions. Contract internal storage variables (e.g., Owner) persist across transactions. The value ranges of internal variables are decided by the transaction sequences. They are not arbitrary because the contract code can only assign specific values to the variables. An example is in Listing 1.

Listing 1: FP caused by $\delta_2$. Reported by VERISMART [8].

1 function init() { fund = 1000; }
2 function withdraw() { uint256 profit = fund*100; } // FP (1000*100 can not overflow)

3 $\delta_3$: Implicit Constraints Between Internal Variables. Smart contracts’ internal storage variables have in-between dependencies. A group of storage variables may always get updated together to keep certain invariants. As shown in Listing 2, the contract variable BALANCE is always equal to pending, so the second function call of a reentrancy attack will never succeed because the first call sends out all of the contract balance.

We tested the state-of-the-art static tools on the currently largest real contract dataset [6]. And then manually checked the result
Listing 2: FP caused by constraint $\delta_3$. Reported by OSIRIS [10] of arithmetic alarms and reentrancy alarms, which are the vulnerability accounts for 95.7% of the contract CVEs [8] and top1 vulnerability in DAGS [5] rank. Table 1 shows the false positives caused by implicit constraints.

Table 1: Implicit constraints results in FPs.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Alarms</th>
<th>FPs</th>
<th>$\delta_1$</th>
<th>$\delta_2$</th>
<th>$\delta_3$</th>
<th>$(\delta_1 + \delta_2 + \delta_3) / \text{FPs}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSIRIS</td>
<td>474</td>
<td>719</td>
<td>79</td>
<td>28</td>
<td>12</td>
<td>88.3%</td>
</tr>
<tr>
<td>VERISMART</td>
<td>106</td>
<td>76</td>
<td>22</td>
<td>17</td>
<td>1</td>
<td>85.9%</td>
</tr>
<tr>
<td>Mythril</td>
<td>[1]</td>
<td>213</td>
<td>125</td>
<td>26</td>
<td>1</td>
<td>23.2%</td>
</tr>
</tbody>
</table>

2 EMPIRICAL EVALUATION

Are the state-of-the-art static vulnerability detectors aware of the implicit constraints? We construct a comparison dataset via bug injection to answer this question. In the control dataset $D_1$, we inject vulnerable code snippets into real contracts (base contracts). All of the vulnerable snippets in $D_1$ are in reachable branches. In the experimental dataset $D_2$, we inject the same vulnerable snippets into the same contracts as $D_1$ but guard each vulnerable snippet with an infeasible condition statement which is opposite to the implicit constraints. Thus, all of the vulnerable snippets in $D_2$ are unreachable.

The more vulnerabilities reported in $D_2$ (n_d2) means the tool missed more implicit constraints. This also indicates the tools have worse abilities in analyzing code accessibility and have more false positives in practice. Taking the vulnerabilities the tools reported in $D_1$ (n_d1) as the baseline, we can calculate the percentage (P) of the implicit constraints the tools can handle ($P = (n_{d1} - n_{d2}) / n_{d1}$).

We insert 7 typical types of vulnerabilities into base contracts with SolidiFI [4] and select three types of contracts as the base contracts: 1) the model contracts which are widely adopted (e.g. ERC20), 2) top contracts [2] which have a large market cap, 3) example contracts from official tutorials. As a result, 973 bugs are injected. More details can be found in the open-sourced repository.

The evaluation result is shown in Table 2. Six out of seven state-of-the-art detectors get similar results in $D_1$ and $D_2$, which means they are not aware of the implicit constraints. Mythril can deal with all three types of constraints by analyzing transaction sequences but suffering from high false-negative rates due to timeout.

3 MITIGATION

We propose a lightweight mitigation method called ConSym for symbolic execution based detectors to deal with implicit constraints and reduce the false positives. It can be easily modeled to the symbolic execution based tools.

For constraint $\delta_1$, ConSym adds constraints to the return value of related instructions (e.g., TIMESTAMP, CALLVALUE) according to their real-world meanings. For example, ConSym limits assets-related variables smaller than 150 million ETH, which is more than the current ETH total supply.

For $\delta_2$ and $\delta_3$, it is not practical to search all of the constraints actively in smart contracts because the search space is very large. Instead, ConSym concretizes such constraints via concolic execution. Firstly, ConSym invokes the contract constructor and functions

---

**REFERENCES**


[7] SunbeamSo. ZEUS evaluation. Retrieved May 9, 2017 from https://docs.google.com/spreadsheets/d/12_g-pkACtCypU3UmTb7ZKtKgM2BG5ErnFi5d-cfTA8


