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 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

ACM Reference Format:
Tingting Yin, Chao Zhang, Yuandong Ni, Yixiong Wu, Taiyu Wong, Xiapu Luo, Zheming Li, and Yu Guo. 2022. An Empirical Study on Implicit Constraints in Smart Contract Static Analysis. In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3510457.3513076

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.

❶ δ_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.

❷ δ_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.

```
function init() { fund = 1000; }
function award() { uint256 profit = fund*100; } //FP (1000*100 can not overflow)
```

Listing 1: FP caused by δ_2. Reported by VERISMART [8].

❸ δ_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

```
function () payable { pending += msg.value; }
function send() {
    owner.depositEth.value(pending)(); // i.e. BALANCE-=pending, FP (reentrancy)
...}
```

Listing 2: FP caused by constraint δ_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 DASP [5] rank. Table 1 shows the false positives caused by implicit constraints.

Table 1: Implicit constraints result in FPs.

Tool	Alarms	FPs	δ_1	δ_2	δ_3	$(\delta_1+\delta_2+\delta_3)$ / FPs
OSIRIS	476	366	204	23	96	88.3%
VERISMART	100*	78	41	17	9	85.9%
Mythril [1]	213	125	26	1	2	23.2%

* Randomly sampled 100 from 2763 alarms for manual verification.

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 D1, we inject vulnerable code snippets into real contracts (base contracts). All of the vulnerable snippets in D1 are in reachable branches. In the experimental dataset D2, we inject the same vulnerable snippets into the same contracts as D1 but guard each vulnerable snippet with an infeasible condition statement which is opposite to the implicit constraints. Thus, all of the vulnerable snippets in D2 are unreachable.

The more vulnerabilities reported in D2 (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 D1 (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[1].

The evaluation result is shown in Table 2. Six out of seven state-of-the-art detectors get similar results in D1 and D2, 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 applied to most of the symbolic execution based tools.

For constraint δ_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 δ_2 and δ_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.*

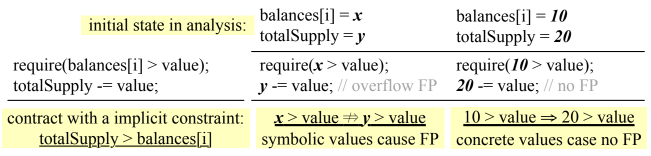
[1] https://github.com/consym/Contract-Constraint-Benchmark

Table 2: Static detectors are insensitive to implicit constraints

	Verifier		Pattern Scanner					
	VERISMART		SECURIFY [11]		smartcheck [9]		slither [3]	
δ_1	🟥		🟥		🟥		🟥	
δ_2	🟥		🟧		🟥		🟥	
δ_3	🟥		🟥		🟥		🟧	
dataset	n_d1	n_d2	n_d1	n_d2	n_d1	n_d2	n_d1	n_d2
reported	139	131	140	126	233	233	430	372
	Symbolic Execution							
	Oyente [7]		OSIRIS		Mythril		ConSym-OSIRIS*	
δ_1	🟥		🟥		🟩		🟩	
δ_2	🟧		🟨		🟩		🟩	
δ_3	🟥		🟥		🟩		🟩	
dataset	n_d1	n_d2	n_d1	n_d2	n_d1	n_d2	n_d1	n_d2
reported	235	209	144	121	51	6	149	8

* OSIRIS with our mitigation ConSym.
* 🟥 P ∈ [0, 20%] 🟧 P ∈ (20%, 40%] 🟨 P ∈ (40%, 60%] ▨ P ∈ (60%, 80%] 🟩 P ∈ (80%, 100%]

Firstly, ConSym invokes the contract constructor and functions with concrete inputs to initialize the internal variables. In this way, the variables will be assigned with concrete values that satisfy implicit constraints. Then, as shown in Figure 1, ConSym can perform symbolic execution with concreted implicit constraints.



initial state in analysis:	balances[i] = ***x*** totalSupply = ***y***	balances[i] = ***10*** totalSupply = ***20***

require(balances[i] > value);
 totalSupply -= value;
	require(***x*** > value);
	 y -= value; // overflow FP
		require(***10*** > value);
		 20 -= value; // no FP

contract with a implicit constraint: totalSupply > balances[i]	$x > \text{value} \not\Rightarrow y > \text{value}$ symbolic values cause FP	$10 > \text{value} \Rightarrow 20 > \text{value}$ concrete values case no FP

Figure 1: Using concrete initialization values to reduce FPs.

We apply ConSym to the open-sourced solution OSIRIS to evaluate it. Table 2 shows that ConSym can deal with all three types of implicit constraints. We further conduct experiments on the real contract dataset [6] to evaluate the performance of the ConSym in terms of false positives and false negatives. The result is shown in Table 3. ConSym reduces the false positives of OSIRIS significantly while not increasing the false negatives.

Table 3: ConSym has much fewer FPs and FNs than OSIRIS.

Tool	Type	False Positive	Positive	False Negative
ConSym	over/underflow	6	133	18
	reentrancy	1	13	0
OSIRIS	over/underflow	158	97	54
	reentrancy	4	13	0

REFERENCES

[1] ConsenSys. 2018. mythril: Security analysis tool for EVM bytecode. https://github.com/ConsenSys/mythril.
[2] Etherscan. 2021. *Etherscan ERC20 top tokens.* Retrieved October 15, 2021 from https://etherscan.io/tokens
[3] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB).
[4] Asem Ghaleb and Karthik Pattabiraman. 2020. How Effective are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection. *arXiv preprint arXiv:2005.11613* (2020).
[5] NCC Group. 2021. *DASP-TOP 10.* Retrieved January 12, 2021 from https://dasp.co/
[6] Kalra. 2017. *ZEUS evaluation.* Retrieved May 9, 2017 from https://docs.google.com/spreadsheets/d/12_g-pKsCtp3lUmT2AXngsqkBGSEoE6xNH51e-of_Za8
[7] melonproject. 2016. oyente: An Analysis Tool for Smart Contracts. https://github.com/melonproject/oyente.
[8] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE, 417–434.
[9] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB).* IEEE, 9–16.
[10] Christof Ferreira Torres, Julian Schütte, et al. 2018. OSIRIS: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference.* ACM, 664–676.
[11] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 67–82.