2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)

VULTRON: Catching Vulnerable Smart Contracts Once and for All

Haijun Wang*, Yi Li*, Shang-Wei Lin*, Lei Ma†, Yang Liu*
*Nanyang Technological University, Singapore. {*haijun.wang,yi_li,shang-wei.lin,yangliu*}*@ntu.edu.sg*
†Kyushu University, Japan. *malei@ait.kyushu-u.ac.jp*

Abstract—**Despite the high stakes involved, smart contracts are often developed in an undisciplined way thus far. The existence of vulnerabilities compromises the security and reliability of smart contracts, and endangers the trust of participants in their ongoing businesses. Existing vulnerability detection techniques are often designed case-by-case, making them difficult to generalize. In this paper, we design general principles for detecting vulnerable smart contracts. Our key insight is that almost all the existing transaction-related vulnerabilities are due to the mismatch between the actual transferred amount and the amount reflected on the contract's internal bookkeeping. Based on this, we propose a precise and generally applicable technique, VULTRON, which can detect irregular transactions due to various types of adversarial exploits. We also report preliminary results applying our technique to real-world case studies.**

I. INTRODUCTION

Smart contracts are computer programs that execute on top of blockchains to manage large sums of money, carry out transactions of assets, and govern the transfer of digital rights between different parties. They provide a mechanism to automatically perform transactions in a trackable and immutable way without an authorized third party. Due to these unique advantages, smart contracts are gaining a lot of popularity and attraction in recent years. Many believe that this technology has the potential to reshape a number of industries, e.g., banking, insurance, supply chains and financial exchange.

Although smart contracts are promising, an increasing number of high-profile attacks resulting in financial losses have been reported recently. Such attacks become possible due to the fact that many smart contracts are developed in an undisciplined way, resulting in vulnerabilities easily exploited by adversaries. One notorious example is the "DAO" attack, i.e., an attacker stole more than 3.5 million Ether (equivalent to about $45 million USD at that time) from the "DAO" contract [1].

Since then, a lot of attempts were made to detect vulnerabilities in smart contracts [1], [2], [3]. However, these techniques all follow a case-by-case manner: contract programs are analyzed and matched with attack patterns observed before. Therefore, the precision and recall of these techniques largely depend on how well the patterns are specified and how comprehensive the pattern collections are. ContractFuzzer [3] proposes seven specific patterns observed in contract programs to detect seven kinds of vulnerabilities. Since the collected patterns are limited and they are modeled at the syntactic-level, the analysis suffers from both false negatives and false positives. For example, the pattern used for detecting inconsistencies in exception handling (so called *exception disorder* [3]) only checks whether functions in a nested call chain throw exceptions. It is considered a vulnerability if the root function does not throw the exception but the subsequent calls do. This produces false positives if all exceptions are properly handled along the call chain. Oyente [4] formulates previously known vulnerabilities as *intra-procedural properties*, and then uses symbolic execution to verify these properties. Similarly, Zeus [5] relies on model checking to discharge the properties derived from previously known vulnerabilities and user-specified policies. A more recently proposed technique, Securify [2], performs a *domain-specific verification* on smart contracts. Again, it assumes that domain patterns for specific types of vulnerabilities are provided in advance. Other works [1], [6], [7] in the same direction all require existing patterns being provided in one way or another.

To overcome these issues, our key insight is that almost all the existing vulnerability patterns (defined at the syntactic-level) can be traced back (semantically) to the mismatch between the actual transferred amount in a transaction and the amount reflected on the contract's internal bookkeeping. Based on this, we propose VULTRON, a precise and general smart contract vulnerability detection technique, which allows to drill down to the very root of the problem. The essence of VULTRON is to build an oracle that can effectively distinguish irregular transactions (usually results of malicious exploits) from normal ones. Our oracle is general enough to cover the previously reported vulnerabilities without the limitation to any particular smart contract programming language or blockchain technology. We demonstrate our approach on Ethereum platform [8], however, the principles behind our approach can be easily adapted and implemented on other platforms. The proposed oracle improves and enables a wide spectrum of downstream analysis techniques, including static analysis such as symbolic execution [1], program verification [5], and dynamic techniques (e.g., testing and fuzzing [3], [7]).

II. TOWARDS A VULNERABILITY DETECTION ORACLE

Vulnerabilities in smart contracts are usually caused by inconsistencies between the implementations and the intended semantics [5]. Transaction-related vulnerabilities can be (most of them have already been) exploited to carry out attacks, e.g., stealing funds from vulnerable contracts. In this section, we discuss the fundamental principles and specific challenges for detecting transaction-related vulnerabilities in smart contracts.

978-1-7281-1758-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-NIER.2019.00009

A. A Simple Contract Model

For demonstration purpose, we consider a simplified model of smart contract. A contract can be abstracted as a tuple $C := \langle a, bal, P, M\rangle$, where $a \in Addr$ is a unique address identifying the contract, $bal \in \mathbb{N}$ is the balance of the contract, $P \in 2^{Addr}$ is a set of account addresses of participants, and $M : P \mapsto \mathbb{N}$ is an internal bookkeeping variable recording the account balances of participants. A transaction $t := \langle s, r, v\rangle$, if performed successfully, deducts v amount from the sending account's balance ($s.bal$ where $s \in Addr$) and transfers the funds to a receiving account at address $r \in Addr$. We denote the values of a variable x before and after the transaction as $pre(x)$ and $post(x)$, respectively.

Note that in Ethereum's case, each contract has a *contract balance* (bal) representing the total amount of funds remaining in the contract, which is out of the contract program's control. Apart from that, contracts managing shared assets also need to keep track of participants' individual account balances, called the *bookkeeping balances*. Contract programs uses an internal *bookkeeping variable* (e.g., `balances` in Fig. 1), which is modeled as M, to record the bookkeeping balances.

B. Balance and Transaction Invariants

Ethereum smart contracts are similar to traditional computer programs in the sense that they are written in a Turing-complete language, e.g., Solidity [9]. However, smart contracts also have very special properties due to their particular usecases. Smart contracts are mainly used to manage the transfer of assets and perform bookkeeping [10]. Furthermore, all contract vulnerabilities which have direct financial implications are due to inconsistencies happened during transactions. Therefore, given that the majority of smart contracts share very specific data structures and underlying business logics, it is possible to come up with common *invariants* for them, whereas this is not possible to do for general purpose computer programs. More specifically, the reasonable smart contracts usually satisfy the following two invariants for their transactions.

Definition 1 (Balance Invariant). *For every contract* $\langle a, bal, P, M\rangle$, $\sum_{p \in P} M(p) - bal = K$, *where* K *is a constant.*

The *balance invariant* requires that the difference between the contract balance and the sum of all participants' bookkeeping balances remain constant, before and after a transaction. This invariant is defined within a single contract, i.e., *intra-contract*, and it ensures the integrity of the bookkeeping balances. If the bookkeeping balances are not updated correctly after a transaction (which violates the balance invariant), then it indicates that an irregular event has happened during that transaction. For example, when an underflow happens during an outgoing transaction, the contract balance naturally goes down while the bookkeeping balances go up instead.

Definition 2 (Transaction Invariant). *For every outgoing transaction* $\langle C.a, r, v\rangle$, $\Delta(M(r)) + \Delta(r.bal) = 0$, *where* $\Delta(x) = post(x) - pre(x)$.

The *transaction invariant* requires that the amount deducted from a contract's bookkeeping balances is always deposited into the recipient's account. This invariant is *inter-contract*, and it is important for ensuring the consistency between the both ends of a transaction. Note that the consistency of incoming transactions can be guaranteed by the balance invariant or other contract's outgoing transaction invariant. In some cases, a transaction may fail in the middle and funds are not transferred, but it may not be captured by the contract's bookkeeping variables, resulting in a vulnerability.

C. Challenges

Now we discuss the challenges involved in materializing the proposed invariants as an oracle and automated tool for detecting irregular transactions due to vulnerabilities.

Identifying Bookkeeping Variables. The key ingredient of both invariants is the bookkeeping variable M. Almost all contracts performing meaningful transactions among multiple parties contain such a variable, usually coming with the name `balances` or `balanceOf`. Some contracts do not have bookkeeping variables, e.g., *King of the Ether* [11]. In these cases, a *ghost variable* and its corresponding updates can be inserted by our analysis. The bookkeeping variable can either be given by the user as an input to our approach or be automatically identified using taint analysis [12]. The idea is to first perform several normal transactions and observe how all the global variables in the contract program change, to find one that always matches with the performed transactions.

Handling Non-Currency Assets. The bookkeeping balances of some contracts may not refer to the crypto-currency directly. This is often the case in ERC20-compliant contracts. In these contracts, participants' digital assets are reflected in terms of the number of available *tokens* rather than `Ether`. These non-currency assets can be converted to `Ether` by multiplying with their prices. The current price of the said assets is stored in some location (e.g., the variable *price*), which can be identified with similar techniques used for locating bookkeeping variables.

Verifying Invariants. The invariants are then translated to program assertions and then be verified either statically or dynamically. The gas consumption of a transaction should also be taken into account in the translation, which is not included in our contract model and invariants. The assertions can be inserted into the compiled Ethereum Virtual Machine (EVM) bytecode, at the end of every function, which can then be checked by existing verification tools [2], [5]. Alternatively, we can instrument the EVM itself to enforce the invariants at runtime. This will prevent irregular transactions from happening even if the deployed contract programs are vulnerable.

III. Vultron on Existing Vulnerabilities

We have implemented a prototype for our approach, and tested it on the Truffle Suite [13]. Then, we review the previously reported vulnerabilities of Ethereum smart contracts and show how our approach can help detect these vulnerabilities.

Reentrancy. The atomicity and sequentiality of transactions may let programmers believe that, when a non-recursive function is invoked, it cannot be re-entered before its termination.

```
 1: contract SimpleDAO {
 2:   mapping (address => uint) public balances;
 3:   function donate(address to) {
 4:     balances[to] += msg.value;
 5:   }
 6:   function withdraw(uint amount) {
 7:     require(balances[msg.sender] >= amount);
 8:     msg.sender.call.value(amount)();
 9:     balances[msg.sender] -= amount;
10:   }
11:   ...
12: }
```

Fig. 1. A simple contract susceptible to the “DAO” attack.

```
1': contract Attacker {
2':   SimpleDAO public dao = SimpleDAO(0x354...);
3':   ...
4':   function () public payable {
5':     dao.withdraw(...);         //Reentrancy
6':   }
7':   ...
8': }
```

Fig. 2. Reentrancy attack on the simple “DAO” contract.

However, this is not always the case, due to the fallback function introduced by Solidity. Take the simplified DAO attack for example. Two contracts, SimpleDAO (the victim, in Fig. 1) and Attacker (in Fig. 2), are deployed on the blockchain. The reentrancy vulnerability of SimpleDAO can be exploited by Attacker as follows. (1) Attacker invokes the donate function of SimpleDAO to donate 1 *wei*, and the balance of SimpleDAO is increased by 1 *wei* automatically and implicitly. Then, the bookkeeping balance of Attacker is increased by 1 *wei* accordingly (Line 4). (2) Attacker invokes the withdraw function to withdraw 1 *wei*. The withdraw function first checks whether the bookkeeping balance of Attacker is sufficient (Line 7), and then transfers 1 wei to Attacker (Line 8). Due to the fallback function mechanism, since Attacker receives money, its fallback function (the one with no name at Lines $4'$–$6'$) will be triggered, and it invokes the withdraw function of SimpleDAO to withdraw money again. Note that balances update at Line 9 is still not executed. (3) The withdraw function is re-entered before the bookkeeping balance of Attacker is updated correctly in its first invocation (Line 9). Thus, in the second invocation, the condition checking at Line 7 still passes, which is not supposed to happen, and the money transfer at Line 8 proceeds, which is not supposed to happen as well. This behavior recursively draws out all the money in SimpleDAO.

The proposed balance invariant, formulated in Definition 1, can help detect the reentrancy attack. Suppose, initially, the balance of SimpleDAO is 10, and the bookkeeping balances of all participants are initialized to zero. Thus, the difference K is 10. After the attack step (1), K is still 10 because the SimpleDAO's balance is increased by 1, so as the bookkeeping balance of Attacker. After the attack step (2), the balance invariant is violated because the SimpleDAO's balance is decreased by 1, but the bookkeeping balance of Attacker does not change, i.e., the difference K becomes 9. Once the proposed balance invariant is violated, we can conclude that the contract is vulnerable.

Exception Disorder. The issue of exception disorder is due to inconsistencies in exception handling. In Solidity, exceptions may be raised in several situations, e.g., the execution runs out of gas, the command throw is executed, etc. However, Solidity is not uniform in handling exceptions. Within a chain of nested calls, there could be two types of exception handling mechanisms [15]: (1) If all the functions in the chain are direct calls, the execution stops and all side effects are reverted, including transfers of Ether. (2) If a function in the chain is a call (the same for delegatecall and send), the exception is propagated along the chain, reverting all side effects, until it reaches the call. From that point on, the execution is resumed with call returning false.

```
 1: contract KotET {
 2:   ...
 3:   function withdraw(address to, uint amount) {
 4:     to.send(amount);          //Gasless Send
 5:   }
 6:   function() {
 7:     ...
 8:     king.call.value(...)(); //Exception Disorder
 9:     king = msg.sender;
10:     ...
11: }
12:}
```

Fig. 3. Example of exception disorder and gasless send.

```
1: contract UnderflowAttack {
2:   ...
3:   function withdraw (uint amount) public {
4:     require(balances[msg.sender] - amount > 0);
5:     msg.sender.transfer(amount);
6:     balances[msg.sender] -= amount; //Underflow
7:   }
8:   ...
9: }
```

Fig. 4. The underflow attack example [14].

Programmers not familiar with the exception mechanism may handle exceptions incorrectly. The transaction invariant, as formulated in Definition 2, can help detect the exception disorder vulnerability. For example, Fig. 3 is a simplified version of “King of Ether Throne”, where Line 8 may cause an exception disorder. In this example, there is no bookkeeping variable and hence we insert a *ghost variable* into the contract. If the return value (Line 8) is used as a branch condition of a if-statement, we update the ghost variable in its $True$ branch. Otherwise, we update the ghost variable immediately after Line 8. Suppose KotET attempts to transfer 10 *wei* to an account r and this transfer fails. The ghost variable (i.e., M(KotET)) is deducted by 10, but r.bal is not increased, which violates the transaction invariant.

Gasless Send. When transferring Ether from one contract to another with the send function, it may end up with an out-of-gas exception. For example, the KotET contract (in Fig. 3) sends Ether to a target contract at Line 4 and this triggers the target's fallback function. If the fallback function contains too many instructions, it may lead to an out-of-gas exception and result in a *gasless send*. If such exception is not handled appropriately, the adversary can keep Ether wrongfully while being seemingly innocent. Our proposed transaction invariant can help detect the gasless send vulnerability by instrumenting with a ghost variable, as in the case of exception disorder.

Integer Overflow/Underflow. Smart contracts primarily operate upon arithmetic operations, e.g., manipulating participants' balances. However, these data are usually strongly typed, and thus their arithmetic operations are susceptible to integer

TABLE I
A COMPARISON OF VULNERABILITY DETECTION TECHNIQUES.

Types	Comparison on Related Techniques				
	Z*[5]	O*[4]	S*[2]	C*[3]	VULTRON
Reentrancy	○	○	○	○	■
Exception Disorder		■	○	○	■
Gasless Send	○	■	○	○	■
Overflow/Underflow	■				■

overflow/underflow. For example, in Fig. 4, the variable `balances[msg.sender]` and `amount` are both unsigned integers. If the variable `balances[msg.sender]` is less than `amount`, the check at Line 4 will pass due to underflow, leading to another underflow at Line 6. Our proposed balance invariant can effectively detect this kind of vulnerability. Suppose the variable `balances[msg.sender]` is 1 and `amount` is 2. After Line 6, `balances[msg.sender]` becomes $2^{256} - 1$, which violates the balance invariant.

To sum up, our approach can detect all the vulnerabilities mentioned above, as summarized in Table I. We use ■ to denote that the technique can precisely detect the vulnerability, ○ to denote that the detection is partial or imprecise, and blank to denote that the technique cannot detect the corresponding vulnerability. Notably, VULTRON precisely detects all variants of the reentrancy vulnerability, the most infamous one in 2018 [14], while all the other techniques may produce false negatives and false positives.

IV. POTENTIAL APPLICATIONS OF VULTRON

The fundamental difficulty when applying traditional security analysis techniques (e.g., testing, verification, monitoring and automated repair) on smart contracts is the lack of general purpose test oracles. This is again attributed to the particular way how smart contracts work – they do not crash and the execution may be silently reverted in cases of irregularities. In this section, we show how VULTRON will enable and improve these techniques when applied to smart contracts.

Testing and Verification. Testing and fuzzing are important dynamic techniques for detecting security vulnerabilities, both of which require a test oracle to determine if a program executes according to the expected behaviors. Similarly, program verification relies on a set of given properties to check if the program aligns with the properties. Coming up with such oracles and properties for general purpose computer programs can be cumbersome and requires significant expertise. For smart contracts, our proposed invariants can easily be translated to oracles and assertions which enable the downstream analyses without additional efforts. We also envision that fuzzing, when equipped with VULTRON, may discover new types of attacks, because our test oracles are general and drill down to the very roots of transaction-related vulnerabilities.

Monitoring and Repair. As mentioned in Sect. II, we can instrument EVM with the appropriate checks to enable the runtime monitoring of contract execution. Upon failures being detected during the execution, we may block the ongoing transactions and even perform repair for the vulnerable contracts. The oracles generated by VULTRON can be used to guide the automatic program repair techniques [16] to construct patches for the detected vulnerabilities.

Beyond Vulnerability. The future of VULTRON is beyond detecting vulnerable contracts. There is another type of contracts which are intentionally designed to take advantage of the participants. We categorize them as the *malicious contracts* (sometimes called *unfair* contracts in the literature [5]). Despite the fact that such contracts are publicly available, most of the participants have no ability to scrutinize and analyze the contract code in order to discover all malicious behaviors in advance. A commonality of such contracts is that some players (usually the contract owner) have significant advantages over the rest of the participants, and this is disguised behind the obscure language syntax and contract logics. VULTRON may be extended to take into account the fairness aspects, which brings analysis of smart contracts to a whole new dimension.

V. CONCLUSION

We propose VULTRON, a general purpose vulnerability detection oracle for smart contracts. Different from previous work, the proposed invariants capture the very roots of transaction-related vulnerabilities, and thus they are not specific to any particular attack pattern. We have demonstrated that these invariants enable to cover the previously reported vulnerabilities. We also believe that our approach can easily be generalized and serves as the driving wheels for a wide range of downstream analysis techniques.

VI. ACKNOWLEDGEMENTS

The authors acknowledge support from grant MOE2018-T2-1-068 (Tier-2, Ministry of Education (MoE), Singapore) and grant NRF2017EWT-EP003-023.

REFERENCES

[1] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang, "sCompile: Critical path identification and analysis for smart contracts," *arXiv*, 2018.

[2] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *CCS*, 2018.

[3] B. Jiang, Y. Liu, and W. Chan, "ContractFuzzer: fuzzing smart contracts for vulnerability detection," in *ASE*, 2018, pp. 259–269.

[4] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *CCS*, 2016.

[5] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *NDSS*, 2018.

[6] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *arXiv*, 2018.

[7] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard: finding reentrancy bugs in smart contracts," in *ICSE*, 2018.

[8] "Ethereum project," https://www.ethereum.org/, accessed 2018.

[9] "Solidity," https://solidity.readthedocs.io/en/v0.4.25/, accessed 2018.

[10] K. Chatterjee, A. K. Goharshady, and Y. Velner, "Quantitative analysis of smart contracts," in *European Symposium on Programming*, 2018.

[11] "King of the Ether," https://github.com/kieranelby/KingOfTheEtherThrone/, accessed 2018.

[12] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: dynamic taint analysis with targeted control-flow propagation." in *NDSS*, 2011.

[13] "Truffle Suite," https://truffleframework.com/, accessed 2018.

[14] "Underflow example," http://www.dasp.co/, accessed 2018.

[15] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (sok)," in *Principles of Security and Trust*, 2017.

[16] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *ICSE*, 2018.