SAFEPAY on Ethereum: A Framework For Detecting Unfair Payments in Smart Contracts

Yue Li* Han Liu* Zhiqiang Yang* Qian Ren* Lei Wang*† Bangdao Chen*
*Oxford-Hainan Blockchain Research Institute
Hainan, China
†Shanghai Jiao Tong University
Shanghai, China

***Abstract*—Smart contracts on the Ethereum blockchain are notoriously known as vulnerable to external attacks. Many of their issues led to a considerably large financial loss as they resulted from broken payments by digital assets, *e.g.*, cryptocurrency. Existing research focused on specific patterns to find such problems, *e.g.*, reentrancy bug, nondeterministic recipient *etc.*, yet may lead to false alarms or miss important issues. To mitigate these limitations, we designed the SAFEPAY analysis framework to find *unfair payments* in Ethereum smart contracts. Compared to existing analyzers, SAFEPAY can detect potential blockchain transactions with feasible exploits thus effectively avoid false reports. Specifically, the detection is driven by a systematic search for violations on *fair value exchange* (FVE), *i.e.*, a new security invariant introduced in SAFEPAY to indicate that each party "fairly" pays to others. The preliminary evaluation validated the efficacy of SAFEPAY by reporting previously unknown issues and decreasing the number of false alarms.**

***Index Terms*—Smart contract; Symbolic execution; Taint analysis; Insecure message call**

I. INTRODUCTION

As smart contracts were introduced on the Ethereum blockchain [1], their security issues have been raising concerns in the ecosystem. Since smart contracts often involve irrevocable payment by real-world digital assets, any vulnerability might lead to permanent financial loss on blockchain. For example, we use a simplified case in Figure 1 to illustrate the DAO attack in 2016 which caused over 50 million US dollars to be stolen on Ethereum. Specifically, the attacker initiated the attack by sending a transaction to the DAO contract to call the `withdraw` function. According to the service logic, a payment will be made from DAO to the sender of the transaction (line 4) with an execution context switch as well. However, the "fallback" function in attacker maliciously calls back to DAO repeatedly to receive more unexpected payment before the balance is updated at line 5.

Ethereum Payment Attacks. Payment attacks as in Figure 1 were classified as *reentrancy bugs* and widely discussed in previous research [2]–[4]. To detect such issues, existing approaches analyze transactions (either statically based on a given contract or dynamically at runtime) to check whether a payment can be reentered according the program context. In addition to reentrancy, other types of payment attacks on Ethereum were studied as well. For example, state update after a payment was used as a pattern to flag potential security problems [5]. A recent work pointed out that nondeterminism due to transaction scheduling can lead to broken payments with manipulated recipients [6]. Unfortunately, heuristics adopted in existing analyzers often introduce false reports in many practical cases. For example, if the comment at line 2 in Figure 1 is removed, the payment at line 4 becomes secure since only authorized accounts are allowed to receive the cryptocurrency. However, existing approaches will still consider the payment as vulnerable thus generate a false alarm.

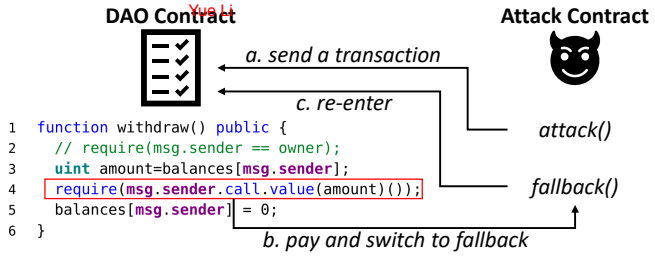

Fig. 1: The simplified DAO attack

Unfair Payment. In this demo paper, we highlighted a new security abstraction for payments in Ethereum smart contracts, *i.e.*, *unfair payment* (UP). Compared to pattern-based heuristics in the literature, UP provides a systematic way to analyze the root cause of a broken payment therefore largely avoid false alarms in the detection. Generally, instead of searching for a certain set of payment-related operations (*e.g.*, payment followed by a write to blockchain state), we reason a complete transaction to check whether a party "fairly" pays to another without making undesired profits. More specifically, the fairness is automatically determined via *fair value exchange* (FVE), which will be explained later. We summarize our main contributions as below.

- We introduced a novel abstraction *unfair payment* (UP), which is more general than existing heuristics.
- We developed the analysis framework SAFEPAY to automatically detect UP in Ethereum smart contracts.
- We conducted a large-scale evaluation in Ethereum and found previously unreported payment problems.

Demonstration Plan. Our demonstration of SAFEPAY will showcase its capabilities in finding real-world security issues of Ethereum smart contracts. The detailed plan includes: i) an

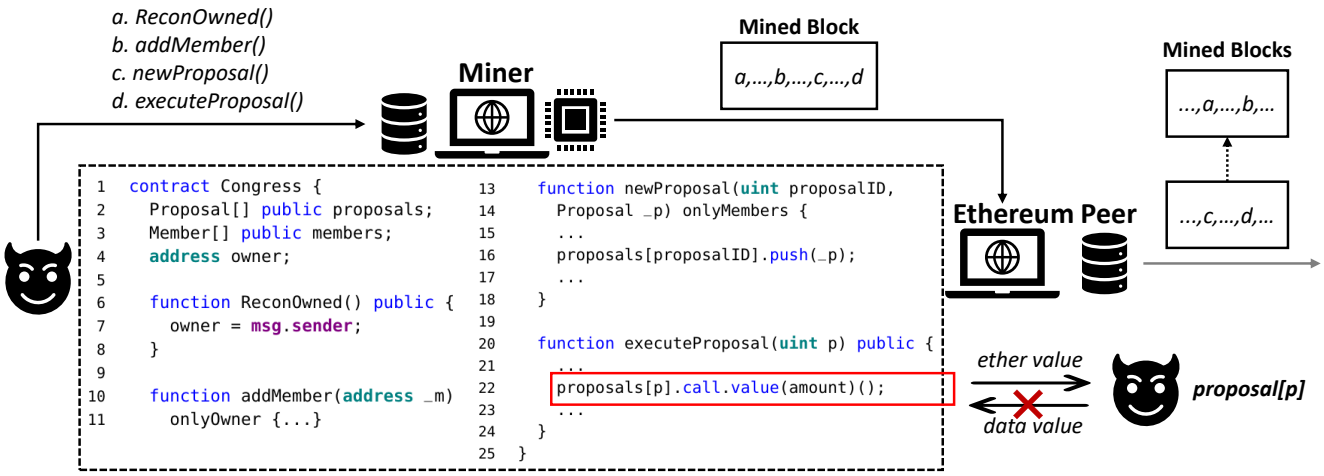

Fig. 2: An example to help illustrate Fair Value Exchange, FVE. Specifically, the mainnet address of the smart contract is `0x5bd6a6d4b21d4b4952426cb23aea7f7bb4944c9b` on Ethereum. We simulated a UP attack on the Ropsten testnet via the transaction `0xcb3198c3dc8f6fb0487aba5ca8fca6364e0db9bcbd004cb632f1eed1cb705faf`.

automatic detection process on a vulnerable contract, ii) in-depth explanation of the detection output and iii) more comparative tests on representative secure/vulnerable contracts. The participants will be able to interact via the SAFEPAY web platform and perform analysis on different smart contracts.

II. FAIR VALUE EXCHANGE

We have designed a new security invariant called *Fair Value Exchange* (FVE) to model the fairness of blockchain payments. A payment m in Ethereum is a 5-tuple $m = \langle g, s, r, v, C \rangle$, where g is the amount of *gas* (*i.e.*, transaction fee) granted for m, s and r are the sender and recipient addresses of the payment respectively, v is the amount of *ether* (*i.e.*, Ethereum cryptocurrency) to be paid and $C = \{c_0, c_1, \cdots, c_n\}$ is a set of conditions to permit the payment. For example, the payment at line 4 in Figure 1 is formulated as $\langle g^*, \texttt{DAO}, I_s, \texttt{amount}, \phi \rangle$. Particularly, g^* denotes the amount of available gas and I_s is `msg.sender`. ϕ is an empty set.

Values on Ethereum. Two types of values are considered on Ethereum, *i.e.*, *ether* and *data*. Specifically, *ether* is the built-in cryptocurrency on Ethereum and *data* refers to the blockchain storage, which is often used to store a specific kind of digital assets, *e.g.*, `ERC20` token.

Value Transfer. Based on two types of values on Ethereum, we now describe the processes of value transfer. To begin with, transfer of ether is realized via payments between two accounts X and Y on blockchain as aforementioned (denoted as $X \mapsto Y$). Moreover, transfer of data value is often fulfilled via updates on blockchain storage, *e.g.*, transfer of an `ERC20` from X to Y (denoted as $X \rightarrowtail Y$) amounts to updating the storage data `balances` in a token smart contract, *e.g.*, $update(\texttt{balances[X]})$ and $update(\texttt{balances[Y]})$. For cases where Y is a smart contract, the value transfer commonly involves only the update on the balance of X itself.

Fair Value Exchange. Conceptually, FVE is designed to model a transaction scenario t (with a single or multiple transactions) where each party fairly "pays" (*i.e.*, transfers values) to others in t. Specifically, given a transaction scenario $t = s_1 s_2 \cdots s_n$ where s_i $(1 \leq i \leq n)$ is an operation, *e.g.*, payment, update on storage, arithmetic operations *etc.*, we use $V^t_{\mapsto}(r)$ to store a key-value table of ether value transfer to a blockchain address r in t. For example, $\langle s, v \rangle \in V^t_{\mapsto}(r)$ indicates a payment with v ether from s to r. Moreover, we use $V^t_{\rightarrowtail}(r)$ to denote a set of data value transfer from r, *e.g.*, $x@i \in V^r_{\rightarrowtail}$ is an update x at program counter i parameterized by r. $\sum_{j=1}^{k}\{v_j \mid \langle s_j, v_j \rangle \in V^t_{\mapsto}(r)\} = 0$ indicates a zero-ether transfer to r. Similarly, $V^t_{\rightarrowtail}(r) = \phi$ means no data values are paid from r. For a blockchain address r, we define an FVE on t *w.r.t.* r if at least one of the following conditions hold: i) $\sum_{j=1}^{k}\{v_j \mid \langle s_j, v_j \rangle \in V^t_{\mapsto}(r)\} \neq 0 \leftrightarrow V^t_{\rightarrowtail}(r) \neq \phi$; ii) t only permits a finite set of addresses to execute. Figure 2 shows an illustrative example. In a transaction scenario with four transactions to a smart contract `Congress`, the attacker enforced a specific ordering (*i.e.*, a, b, c, d) to (a) lift himself as an owner, (b) add himself as a member, (c) insert a new proposal and (d) make a payment from the contract to himself. The four transactions were packaged into a block in the process of mining. Given different values of gas, the specific ordering can be accepted in other Ethereum peers with a large probability. The transaction scenario violated FVE by involving an ether transfer to `proposal[p]` but without any data value transfer, therefore led to an UP.

III. THE SAFEPAY FRAMEWORK

Based on the FVE invariant as introduced above, we designed the SAFEPAY framework to automatically detect unfair payments in Ethereum smart contracts. The architecture of SAFEPAY is shown in Figure 3. Generally, four public interfaces are provided for users to integrate SAFEPAY with their own software. *request* and *response* are designed to

start security analysis and retrieve output. *configure* and *query* are used to specify and check functional parameters. Given an input smart contract (either with Solidity source code or EVM bytecode), the *Input Preprocessor* is executed to produce necessary information for further analysis, *e.g.*, an Application Binary Interface (ABI) file, the assembly code of the contract and a source map file. Next, a *CFG Builder* is called to generate the Control Flow Graph (CFG), *i.e.*, group instructions into connected basic blocks. Particularly, the CFG in this step might be incomplete since some basic block transitions are known at runtime. Then, CFG is used as a representation of the contract in following processes.

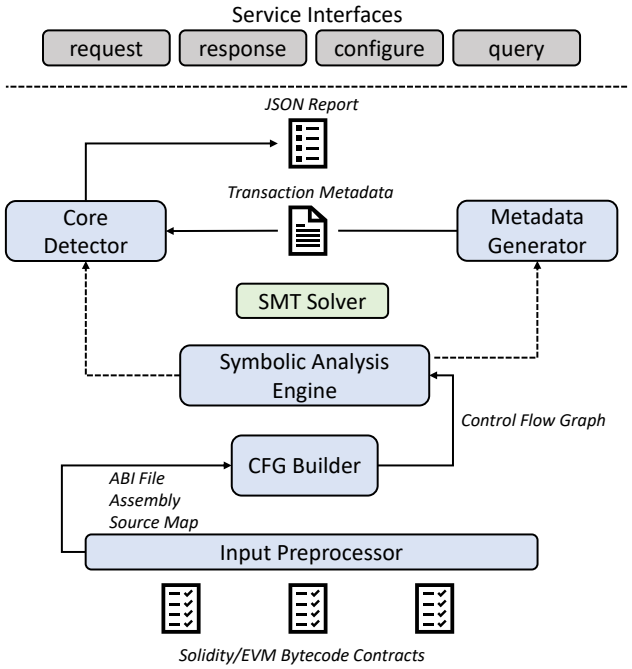

Fig. 3: The architecture design of SAFEPAY.

Symbolic Analysis Engine. Security analysis in SAFEPAY is started from the module *Symbolic Analysis Engine*, which is designed to symbolically execute possible transactions on the given contract CFG and adopted in previous research [2], [7]. The process in SAFEPAY is similar and works as follows. First, we symbolize a transaction by using symbolic values for its data, *e.g.*, transaction input is represented as the symbol I_d. Then, basic blocks from a CFG are iteratively fetched for execution on the symbolic input. For each basic block, all the instructions are interpreted by the engine to update contract storage or communicate with other contracts on blockchain. Unlike traditional processes, SAFEPAY keeps track of data flow in this step to capture important data dependencies. This is realized via *taint analysis* [8], *i.e.*, mark the transaction input as a taint and propagate tainted values during execution. For example, on symbolically executing the `ReconOwned` function in Figure 2 (line 6-8), taint analysis would create a dependency from `msg.sender` (address of the transaction sender) to the storage data `owner`. After the execution of a basic block completes, a following block is selected to execute. Particularly, the engine generates a path condition to reach the new block, *i.e.*, a constraint on symbolic values. An SMT solver, *e.g.*, Z3[1], is used to solve the constraint and skip infeasible transactions. The symbolic execution works in a systematic manner to refine the CFG on the fly and cover it as much as possible.

Metadata Generator. During a CFG of a contract is being symbolically executed, a *Metadata Generator* is repeatedly invoked to produce transaction metadata for security analysis. Given a transaction tx, its metadata S_{tx} is $S_{tx} = \langle U_{tx}, P_{tx} \rangle$ where U_{tx} is a set of conditional updates on storage (*i.e.*, data value transfer) and P_{tx} is a set of payments (*i.e.*, message calls [1] with or without ether attached). For a storage update $u : \langle v, l, \{c_0, \cdots, c_k\} \rangle \in U_{tx}$, v and l denote the new value v of the storage at location l. $\{c_0, \cdots, c_k\}$ stands for a set of conditions to reach u. Moreover, for a payment $p : \langle g, s, r, v, C \rangle @i \in P_{tx}$, it is encoded as described in §II with a program counter i. In general, *Metadata Generator* offers interfaces to other components, *e.g.*, *Symbolic Analysis Engine*, to generate transaction metadata. In this sense, the generation of metadata is decoupled from symbolic execution such that a new type of metadata can be flexibly extended. Once a transaction in CFG is processed, the corresponding metadata is produced and sent to the *Core Detector* to search for potential unfair payments.

Core Detector. The detection of UP is performed by the *Core Detector* in SAFEPAY, which takes as input a group of transaction metadata and produces a JSON report for UP issues. Similar to *Metadata Generator*, *Core Detector* is invoked by *Symbolic Analysis Engine* at runtime. The algorithm to detect UP works as follows:

Step 1 Generate a set T of transaction scenarios with a specific bound k. For $t \in T$, $t = tx_1 tx_2 \cdots tx_k$ has no more than k transactions and at least one payment.

Step 2 Pick a $t = tx_1 tx_2 \cdots tx_m$ from T and analyze its sequence of transaction metadata $S_1 S_2 \cdots S_m$ ($m \leq k$).

Step 3 For $S_i = \langle U_i, P_i \rangle$ ($1 \leq i \leq m$), we update the $V^t_{\mapsto}(r)$ and $V^t_{\rightarrow}(r)$ according to value transfers involved in U_i and P_i for a specific address r.

Step 4 Once all the metadata in t is processed, we detect UP in t according the invariant defined in §II. If at least one UP is detected, we store t to the report.

Step 5 If T is fully covered, the detection completes. Otherwise, go back to **Step 2**.

IV. PRELIMINARY EVALUATION

The preliminary evaluation for SAFEPAY was conducted on two open-sourced security benchmarks[2][3], as shown in Table I. We only considered payment-related issues. Comparison analyzers include Securify [5], Mythril[4] and Oyente [2].

[1]https://github.com/Z3Prover/z3
[2]https://github.com/SmartContractSecurity/SWC-registry
[3]https://github.com/crytic/not-so-smart-contracts
[4]https://github.com/ConsenSys/mythril-classic

TABLE I: UP Detection results. ✓indicates a true positive or negative. ✗ refers to a false positive or negative.

Contract	SAFEPAY	Securify	Mythril	Oyente
simple_dao	✓	✓	✓	✓
DAO	✓	✓	✗	✓
SpankChain_Payment	✓	✗	✗	✗
modifier_reentrancy	✗	✗	✓	✗
simple_dao_fixed	✓	✓	✗	✓
modifier_reentrancy_fixed	✓	✓	✗	✓

Compared to existing analyzers, SAFEPAY was more accurate on six representative cases (the first four are buggy and the last two are non-buggy). We have also conducted a large-scale evaluation on deployed Ethereum smart contracts, where 46,237 contracts were considered in total. In terms of accuracy, over 70% reports from comparison analyzers were false positives while only 11.9% for SAFEPAY. There were also cases where SAFEPAY reported UP problems that were missed by others. Figure 2 was a case in this kind. We do not include all the details due to space limit.

V. DEMONSTRATION DESCRIPTION

Currently, the SAFEPAY framework is implemented in Python as a command line tool for Linux-based operating systems, which requires dependencies including Python 2.7, the `solc` compiler for Solidity[1] and `evm` for Ethereum. A simple command to run SAFEPAY is:

```
$ safepay -s example.sol -o report.out -src -b 2
```

Specifically, `safepay` is a running script to start a detection. `-s` and `-o` specify the input and output files respectively. `-src` indicates the input is a source code contract (`-bin` for an input of binary code). `-b` is the value of analysis bound as described in §III. In this case, we configured SAFEPAY with a bound of 2, *i.e.*, SAFEPAY would consider transaction scenarios with no more than 2 transactions. In the demonstration, we set up a web-based platform with SAFEPAY as a back-end for users to conveniently interact with the framework. Two use cases will be used to demonstrate SAFEPAY, *i.e.*, basic detection and analysis tuning via the bound configuration. While the first use case is to help participants understand basic capabilities of SAFEPAY, the second use case is designed to show useful skills when analyzing complicated smart contracts.

Use Case 1: Basic Detection. In the first use case, the participants will be using SAFEPAY on two given smart contracts, *i.e.*, a vulnerable contract with known UP and a secure contract, respectively. First, we go through the source code of the vulnerable contract with participants and introduce its code structures. Next, a participant will start the detection of SAFEPAY using the basic configuration. During the analysis, we let him or her monitor the runtime logs generated at back-end of the web platform. When the analysis finishes, the platform will display an overview of the detection, *e.g.*, the number of vulnerabilities found, time used in the detection *etc.*. Furthermore, we go into details of the detected UP with participants and explain how the potential attack manages to exploit an unfair payment. Furthermore, the participants can try simulating the attack on Ethereum testnet by creating malicious transactions based on the detection report that is just generated by SAFEPAY.

Additionally, the participants will be asked to fix the vulnerable contract by modifying the buggy code. Once done, they can put the fixed contract to SAFEPAY again for regression analysis. In this run, the participants can make sure their fix is correct if no more UP is reported by SAFEPAY.

Use Case 2: Analysis Tuning. In further, the second use case in the demonstration will be focusing on analyzing more complicated smart contracts with SAFEPAY. Participants will be given a group of medium-size and large-size contracts for UP detection. Then, they will be guided to use different bound values to analyze those contracts. More specifically, participants will be able to learn about the tradeoff in tuning SAFEPAY. For example, by understanding the application context of a contract (*e.g.*, what kind of transactions the contract may receive), would we be able to bound the analysis in a cost-efficient manner. For example, a smart way to use SAFEPAY for library contracts is specifying a small bound value (*e.g.*, 1 or 2), because those contracts commonly receive similar transactions. In contrast, we could try relatively large bound values (*e.g.*, 3 or 4) for important service contracts, which often communicate with the blockchain under various contexts, *e.g.*, distributed and dependent transactions whose schedule is highly non-deterministic. In that case, a large bound value can help dig out hidden payment issues although it may be more expensive to run SAFEPAY.

[1]https://solidity.readthedocs.io

VI. ACKNOWLEDGEMENT

Lei Wang is supported by National Key Research and Development Program of China (No.2018YFB0803400, No.2019YFB2101601).

REFERENCES

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[2] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.

[3] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 65–68.

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

[5] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 67–82.

[6] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[7] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1317–1333.

[8] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.