Protect Your Smart Contract Against Unfair Payment

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

***Abstract*—While smart contracts have enabled a wide range of applications in many public blockchains, *e.g.*, Ethereum, their security issues have been raising an increasing number of threats on the stability of blockchain ecosystem. In practice, many external attacks on smart contracts result from broken payments with digital assets, *e.g.*, cryptocurrencies. While an increasing number of research works have been focusing on such problems, many of them adopted pattern-based heuristics (*e.g.*, reentrancy) to find payment-related attacks thus can incur a considerably large portion of both false positives and negatives.**

To overcome these limitations and achieve better payment security on blockchain, we introduced a new class of payment attacks in this paper, *i.e.*, *unfair payment* (UP). Compared to existing heuristics, UP semantically captures a wider range of payment attacks. Furthermore, we highlighted the general framework SAFEPAY to systematically detect UP. The key insight behind is a novel security invariant, *i.e.*, *fair value exchange* (FVE), which models the fairness for blockchain payments between multiple parties. More specifically, SAFEPAY systematically explores the transaction space of a given smart contract and generates a bounded set of transaction sequences. For each of the sequence, SAFEPAY reports a UP attack once a violation on FVE is confirmed. We have further instantiated SAFEPAY for Ethereum and applied it in real-world smart contracts. In the empirical evaluation, SAFEPAY managed to identify previously unreported UP attacks and effectively avoid false alarms compared to analyzers in the literature as well.

***Index Terms*—Smart contract; Unfair payment; Fair value exchange**

I. INTRODUCTION

Smart contracts were introduced by Ethereum [1] to perform transparent and traceable transactions on blockchain. While they are able to enable many complicated business services, *e.g.*, cryptocurrency, insurance, supplychain *etc.*, they are prone to various forms of security attacks. Even worse, since smart contracts are often associated with real-world assets, any contract vulnerability may lead to catastrophic consequences, *e.g.*, money loss, financial disorder *etc.* For example, the notorious DAO attack resulted in a loss of over 50 million US dollars. We use a simple contract to briefly explain the DAO attack, as shown in Figure 1. Specifically, the contract is written in the Solidity language [2]. The attacker initiated the attack by sending a transation to call the `withdraw` function. According to the service logic, a payment of *ether* (*i.e.*, cryptocurrency on Ethereum) will be made from TokenBox to the sender of the transaction though a message call 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 `balances` is updated at line 5.

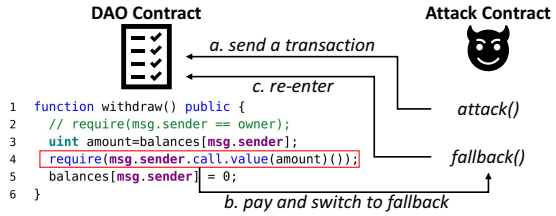

Fig. 1: The simplified DAO attack

Ethereum Payment Attacks. Payment attacks like this were classified as reentrancy bugs and widely discussed in previous research [3]. From the practical perspective, reentrancy bugs in Ethereum can be easily exploited by external attackers. To detect such security issues, existing approaches have leveraged program analysis techniques to find specific patterns [3]–[5]. Commonly, such a pattern would 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 [4]. 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 are often imprecise in many practical cases. For example, if the sanity check at line 2 in Figure 1 is uncommented, 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. There might be other cases where payment attacks are missed, leaving a smart contract at risk.

Unfair Payment Attack. In this paper, we have revisited the state-of-art payment analyses and highlighted a new class of payment attacks on smart contracts, *i.e.*, *unfair payment* (UP). As a security abstraction, UP captures a wider range

† Corresponding author.

2575-8462/20/$31.00 ©2020 IEEE
DOI 10.1109/SRDS51746.2020.00014

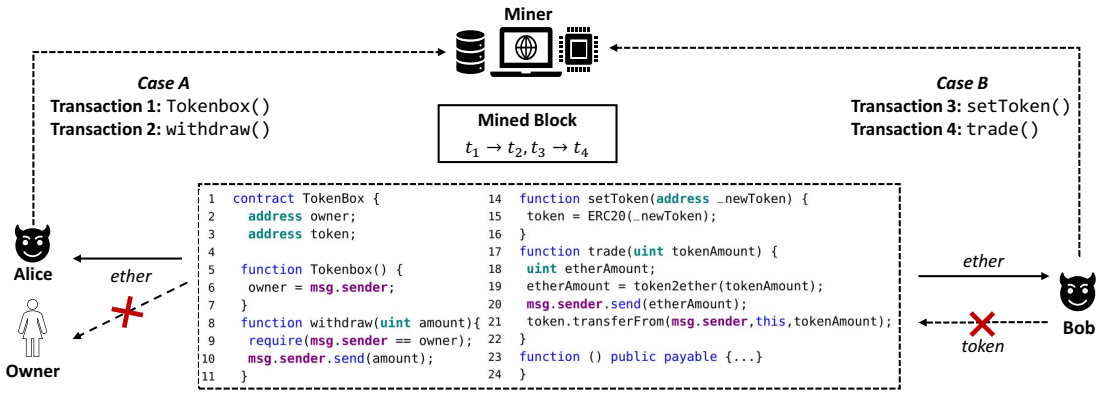

Fig. 2: Common programming patterns related to UP

of practical threats compared to existing types of attacks. To better understand UP, we use the illustrative example below to explain its characteristics. Figure 2 shows an Ethereum contract called `TokenBox`, which serves as a trading platform for cryptocurrency tokens. External accounts are allowed to trade a specific type of tokens for *ether*. Specifically, `TokenBox` creates two *state variables* (`owner` and `token`), whose values are permanently stored on blockchain. Besides, the contract defines four functions and an unnamed *fallback* function to process external transactions.

Case A: Free owner for everyone. The function `withdraw` (line 8-11) allows the owner of `TokenBox` to withdraw funds (*ether*). However, due to a typo in the function name ("`Tokenbox`" instead of "`TokenBox`"), `Tokenbox` (line 6-8) works as a normal public function rather than the contract constructor which is called only when the contract is initiated [2]. We further describe a specific case with two transactions involved where any external attackers (*e.g.*, Alice and Bob) could make him/herself the owner.

Transaction 1 Alice calls `Tokenbox` to lift herself up to an owner (line 6).

Transaction 2 Alice `withdraw()` a specific amount of *ether*.

With transaction 2 following 1 in a mined block, an attacker Alice would be able to bypass the check at line 9 and perform a one-way payment from the contract to arbitrary accounts, leading to a UP.

Case B: Costless trade for ether. The function `trade` (line 25-30) enables a token swap service between `TokenBox` and another account (*i.e.*, `msg.sender`). Specifically, `msg.sender` gets *ether* from `TokenBox` (line 20) and pays a specific type of tokens to `TokenBox` (line 21). Consequently, an attacker Bob could exploit a UP at line 21 via two steps.

Transaction 3 Bob creates an externally-controlled address `token` by calling `setToken()` (line 15-17).

Transaction 4 Bob recursively enters `trade()` when the execution is given to `token` at line 21.

In this specific case of UP, there are two payments (line 20 and 21) where a reentrancy could be manifested in one of them (when `transferFrom` at line 21 is not defined at the address of `token`)*. That way, the contract would be iteratively paying to `msg.sender` without receiving anything.

Particularly, both of the payments in ***Case B*** are considered secure against reentrancy attacks as discussed in [3], [4], [7], because they are neither re-enterable (line 20) nor associated with ether (line 21). Yet combined together, they lead to an unfair payment situation. Although UP provides a general view on security attacks of smart contracts, they are hard to detect. We summarize the main challenges below.

Challenge 1: General Definition of UP. There is no general security definitions for unfair payments at the time of writing, making it difficult to design systematic analyses and tools to find such payment attacks in practice.

Challenge 2: Transaction Space Exploration. Finding UP requires analyzing transaction scenarios with possibly complicated control-flow and data-flow, *e.g.*, track a payment recipient across multiple transactions. Such analysis is essentially hard due to a potentially large transaction space.

Challenge 3: Payment Analysis. A key task of the UP detection is to understand payment in smart contracts, *i.e.*, who is paying to whom. However, given the fact that there are different types of blockchain payments, a general way to analyze payments is highly desired.

Our Insight. To address these challenges, we highlighted a novel analysis framework SAFEPAY to detect UP issues in smart contracts. The key insight behind is a new security invariant, *i.e.*, *fair value exchange* (FVE), which models the fairness of blockchain payments. Compared to existing pattern-based heuristics, the proposed invariant is able to identify hidden vulnerable scenarios without incurring too many false reports. Based on FVE, a UP is flagged if a violation on FVE is found. To this end, SAFEPAY systematically explores the

*Fallback function is called if the target function cannot be found [1].

transaction space of a given smart contract by symbolically executing it and generating feasible transaction sequences within a specific bound. Furthermore, we have instantiated SAFEPAY and applied it in finding UP in Ethereum smart contracts. In the evaluation, SAFEPAY managed to report previously unknown payment attacks and avoid a large portion of false alarms compared to analyzers in the literature.

Contribution. Main contributions of this work are as below.

- We introduced the *unfair payment* (UP) attack of smart contracts which is semantically different from existing attacks and further highlighted *fair value exchange* (FVE) as a formal security invariant on blockchain payments.
- We have designed the SAFEPAY analysis framework based on FVE to systematically detect potential UP attacks in a given smart contract. Compared to existing analyses, SAFEPAY is able to achieve a better balance between false negatives and positives.
- We have conducted a large-scale evaluation in Ethereum and found previously unreported payment attacks.

Paper Organization. The remainder of this paper is organized as follow. §II introduces background information of Ethereum, Ethereum Virtual Machine and blockchain transactions. In §III, we present the design of SAFEPAY. §IV shows the empirical evaluation results on SAFEPAY. §VI discusses on related works. §VII concludes the whole paper.

II. BACKGROUND

A. Ethereum Assets

Ethereum has two types of assets called *ether* and token. *ether* as Ethereum cryptocurrency is the fundamental asset for operation of Ethereum, which could be transferred between accounts through two types of message calls. The first type of message call implements *ether* payment with high-level programming APIs. There are several built-in APIs in Solidity [2], *e.g.*, including `call.value()()`, `send()`, `transfer()` *etc.* Another type of message call is to invoke a normal external contract function with `payable` tag. The `payable` in Solidity enables the function to receive *ether* into the contract from the caller. The token within Ethereum began when someone wrote a smart contract to manage balances which are actually value counters stored in a contract. For example, the `balances` in Figure 1 at line 3. It is literally a mapping of addresses to numbers storing the balance of each address. The token transfer occurs on the update of local storage (line 5 in Figure 1) or the update of storage in a token-managed contract via a message call (line 21 in Figure 2). One of the most significant contracts for token-management is known as `ERC20`, which has emerged as the technical standard used for smart contracts to design tokens. Specifically, the `transferFrom` function in `ERC20` takes three arguments, *i.e.* `from`, `to` and `value`, which performs the update of `balances[from]` and `balances[to]`.

B. Ethereum Virtual Machine

Ethereum Virtual Machine (EVM) runs smart contract codes to automatically conduct transactions on Ethereum blockchain [8]–[10]. The Ethereum Virtual Machine has three locations where it can store data: stack, memory and storage. The Storage, a persistent memory area for each account, contains all the contract state variables, for instance, the state variable `owner` in line 2 of Figure 2. The Memory is used to hold temporary values, for instance, the variable `etherAmount` in line 19. The Stack is used to hold small local variables and perform all calculations. EVM is designed with its own set instructions, which supports arithmetic, bit, logic and comparison operations. We informally explain 3 types of instruction closely related to UP. The message call is triggered by `CALL` instruction. `CALL` specifics 7 parameters, *i.e.*, gas value given for the call, callee address, ether value attached to the call, input offset, input length, output offset and output length. This callee address can be derived from the caller address (`msg.sender`), input data, or storage variables. `SSOTRE/SLOAD` stores value and loads value from the storage respectively. `JUMP/JUMPI` causes a jumping operation from current instruction to a specific offset.

III. DESIGN OF SAFEPAY

A. Fair Value Exchange

§I informally explained FVE, now we describe its formal definition. To begin with, an *ether* payment operation is formulated as $m = \langle g, s, r, v, C \rangle$. Specifically, g is the maximum amount of *gas* (*i.e.*, transaction fee in Ethereum) granted for m. s, r is the address of the sender and recipient, v is the amount of *ether* (*i.e.*, Ethereum cryptocurrency) to be paid in m, $C = \{c_0, c_1, \cdots, c_n\}$ is a set of path conditions to permit the payment. For example, the payment at line 4 in Figure 1 is formulated as $\langle g^*, \texttt{DAO}, I_s, \texttt{amount}, \phi \rangle$. g^* is the amount of gas given to the transaction on `withdraw`, the sender of payment is `DAO`, I_s is represent `msg.sender` which is the recipient of payment, amount is value of *ether* to be paid and ϕ is an empty set.

Definition 1 (Ethereum Value)**.** We define the value of Ethereum in two forms, *i.e.*, *ether* and *token* respectively. While *ether* is the cryptocurrency on Ethereum, *token* is often implemented as a specific storage data.

Definition 2 (Value Transfer)**.** A contract value transfer is a triple $\langle X, Y, \rightarrow \rangle$, where X and Y are the address of the payer and payee respectively. $\rightarrow \in \{\rightarrow_e, \rightarrow_t\}$ denotes the transfer operations on Ethereum values. There are two types of allowed operations, *i.e.*, direct *ether* transfer ($\rightarrow_e$) between X and Y, and *token* update ($\rightarrow_t$) on the blockchain storage. For example, transfer of an `ERC20` from X to Y (denoted as $X \rightarrow_t Y$) amounts to updating the storage data `balances` in a token smart contract, *e.g.*, $update(\texttt{balances[X]})$ and $update(\texttt{balances[Y]})$.

Definition 3 (Fair Value Exchange)**.** FVE is designed to model a transaction scenario t (with single or multiple transactions)

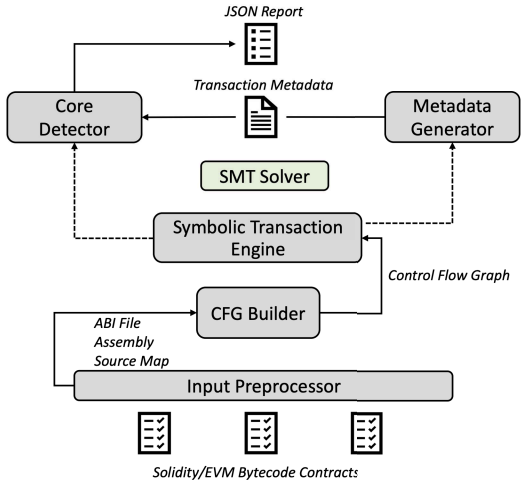

Fig. 3: The general workflow of the SAFEPAY framework

where each party fairly pays to others. For an address r and a transaction scenario $t = s_1 s_2 \cdots s_n$ where s_i $(1 \leq i \leq n)$ is an operation, *e.g.*, ether payment, update on storage *etc.*, we use $V^t_{\rightarrow_e}(r)$ to store a key-value table of ether value transfer to a blockchain address r in t. For example, $\langle s, v\rangle \in V^t_{\rightarrow_e}(r)$ indicates a payment with v ether from s. Moreover, we use $V^t_{\rightarrow_t}(r)$ to denote a set of token transfer from r, *e.g.*, $x@i \in V^r_{\rightarrow_t}$ is an update x at program counter i parameterized by r. $\sum_{j=1}^{k}\{v_j \mid \langle s_j, v_j\rangle \in V^t_{\rightarrow_e}(r)\} = 0$ indicates a zero-ether transfer to r where k is the length of $V^t_{\rightarrow_e}(r)$. Similarly, $V^t_{\rightarrow_t}(r) = \phi$ means no data values are paid from r. For a blockchain address r, we said that t is an FVE with a specific bounds k if at least one of the following conditions hold:

i) $\sum_{j=1}^{k}\{v_j \mid \langle s_j, v_j\rangle \in V^t_{\rightarrow_e}(r)\} \neq 0 \leftrightarrow V^t_{\rightarrow_t}(r) \neq \phi$;

ii) t only permits a finite set of addresses to execute, *e.g.*, the `owner` of the contract.

B. Overview

Based on the FVE invariant, we designed the SAFEPAY framework to automatically detect UP. SAFEPAY takes as input a smart contract p to be analyzed, either in the form of Solidity source code or EVM bytecode. The *Input Preprocessor* is executed to produce Application Binary Interface (ABI) file, the assembly code of the contract and a source map file for further analysis. The *CFG Builder* constructs the control flow graph (CFG) $\mathcal{G}_p$ for p. This is done via generating basic blocks and connecting them based on the `JUMP` and `JUMPI` instructions. At this point, $\mathcal{G}_p$ may have unconnected basic blocks for the reason that some of the jump target addresses cannot be resolved statically. Then, the *Symbolic Transaction Engine* performs symbolic execution process on p with symbolic transaction variables, *e.g.*, sender address, input data, block information *etc.*. $\mathcal{G}_p$ is iteratively refined in this step by connecting isolated basic blocks. Furthermore, *Symbolic Transaction Engine* tracks the transaction dataflow using a dynamic taint analysis technique (§III-C) so that we know how different values (especially storage data) are dependent on specific *taints*, *e.g.*, first four bytes of transaction input. After a program path t in $\mathcal{G}_p$ is executed, *Metadata Generator* generates *transaction metadata* for t (§III-D). Generally, the metadata captures two types of information of t, *i.e.*, path conditions and state updates. Based on all the collected metadata in $\mathcal{G}_p$, The *Core Detector* employs an SMT solver to detect UP in p (§III-E). Particularly, the detection aims at checking satisfiability of the FVE oracle as defined in §III-A for not only a single transaction scenario but multiple transactions as well. If a UP is found, SAFEPAY will produce a JSON report for UP issues to help developers understand the problem.

C. Symbolic Transaction Engine

Security analysis in SAFEPAY is started from the module *Symbolic Transaction Engine*, which is designed to symbolically execute possible transactions on the given contract CFG and adopted in previous research [3], [11]. The process of *Symbolic Transaction Engine* is similar and works as follows. Firstly, each symbolic path in the CFG is referred to a single transaction, 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. All the instructions in basic block are interpreted by the engine to update contract storage or communicate with other contracts on blockchain.

Unlike traditional processes, we track the important dataflow information on the fly and compute the important data dependency relationship based on it. Specifically, we choose to use a dynamic taint analysis technique that marks *taints* and tracks their flows in the execution [12]. In our context, such taints include three types of sources $\{I_s, I_d, H\}$. I_s and I_d is the sender of a transaction and the transaction input data, respectively. H represents block variables [1], *e.g.*, block number, block hash *etc.* In addition, we use the symbols I^*_s, I^*_d and H^* to represent values that are dependent on $\{I_s, I_d, H\}$ respectively. To track the flow of these taints, SAFEPAY propagates tainted data in different spaces (*i.e.*, stack, memory, storage) when symbolically executing an EVM instruction. More specifically, the propagation rules are defined in Table I.

For example, the EVM instruction `CALLDATALOAD` gets 32 bytes of transaction input data $I_d(i : i+32)$ and puts them on the stack. SAFEPAY extends the structure of stack frames and uses a special field to represent a taint I^*_d from I_d. Similarly, other types of taints will be tracked in this way when SAFEPAY interprets an instruction. By checking the taint field of a specific data space, *e.g.*, stack frame, we would be able to know whether the data is dependent on any taint source. We use the following example for further explanation.

Example. Figure 4 shows a partial control-flow graph (CFG) of the `withdraw` function in Figure 2. We also highlight the EVM stack to explain the process of taint propagation as described in Table I. Specifically, the transaction input I_d is marked as a taint and pushed onto the stack as `CALLDATALOAD`

TABLE I: Representative taint propagation rules in SAFEPAY

Instrcutions	Before	After	Description
`CALLDATALOAD` i	$[\langle\bot,\bot\rangle,\cdots]$	$[\langle\bot,\bot\rangle,\langle I_d(i:i+32),I_d^*\rangle,\cdots]$	Taint from transaction input
`CALLER`	$[\langle\bot,\bot\rangle,\cdots]$	$[\langle\bot,\bot\rangle,\langle I_s,I_s^*\rangle,\cdots]$	Taint is the sender of transaction
`CALLDATACOPY` m,i,l	$m[\langle\bot,\bot\rangle,\cdots]$	$m[\langle\bot,\bot\rangle,\langle I_d(i:i+l),I_d^*\rangle,\cdots]$	Copy transaction input to memory
`SSTORE` s,i	$s[\langle\bot,\bot\rangle,\cdots]$	$s[\langle\bot,\bot\rangle,\langle I_d(i:i+256),I_d^*\rangle,\cdots]$	Writes a value to storage
`NUMBER`	$[\langle\bot,\bot\rangle,\cdots]$	$[\langle\bot,\bot\rangle,\langle H.num,H^*\rangle\cdots]$	Taint form block number
`BLOCKHASH` $H.num$	$[\langle\bot,\bot\rangle,\cdots]$	$[\langle\bot,\bot\rangle,\langle Hash(H.num),H^*\rangle\cdots]$	Taint form the hash of block number
`ADD` v_1 v_2	$[\langle\bot,\bot\rangle,\langle v_1,I_d^*\rangle,\langle v_2\rangle\cdots]$	$[\langle\bot,\bot\rangle,\langle v_1+v_2,I_d^*\rangle,\cdots]$	Taint propagation through `ADD`
`EQ` v_1 v_2	$[\langle\bot,\bot\rangle,\langle v_1,I_d^*\rangle,\langle v_2\rangle\cdots]$	$[\langle\bot,\bot\rangle,\langle computed,I_d^*\rangle,\cdots]$	Taint propagation through `EQ`

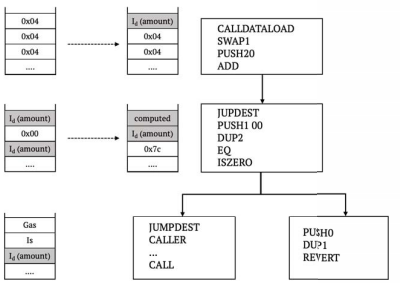

Fig. 4: Partial control-flow graph of the `withdraw`

is symbolically executed (top). When an `EQ` instruction is executed on the top two stack elements, *i.e.*, I_d and `0x00`, the symbolic result `computed` is tainted as well since its value is dependent on I_d (middle). As the symbolic transaction proceeds to the `CALL` instruction (the payment at line 10 in Figure 2), a tainted I_d is the top third stack element (bottom). That said, the I_d is used as a parameter of the payment instruction `CALL`. In this manner, we are able to track the data flow dependency through a symbolic transaction according to the taint propagation rules defined in Table I.

D. Metadata Generator

In the symbolic execution process of SAFEPAY, a *Metadata Generator* is repeatedly invoked to produce transaction metadata for security analysis. As described in §III-B, each transaction tx in this process is modeled via *transaction metadata*, *i.e.*, path conditions of tx and state updates in tx. More formally, a metadata S_{tx} of tx is defined as a tuple $S_{tx} := \langle U_{tx}, P_{tx}\rangle$, where U_{tx} is a set of token payment (*i.e.*, updates on local storage or external storage), and P_{tx} is a set of *ether* payment (*i.e.*, message call [1] with ether attached) in t. For a local storage update $u_l : \langle v, l, \{c_0, \cdots, c_k\}\rangle \in U_{tx}$, 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_l. And the external storage update via the message call is defined as $u_e : \langle t, m, \{c_0, \cdots, c_k\}\rangle \in U_{tx}$, where t is the address of external contract, m is a set of address type variables that may involve external storage updates which captured by analyzing the input data of the message call. Moreover, for an *ether* payment $p : \langle g, s, r, v, C\rangle@i \in P_{tx}$, it is encoded as described in §III-A with a program counter i. In general, *Metadata Generator* offers interfaces to *Symbolic Transaction 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.

Example. Consider function `withdraw` in Figure 3. The transaction of it only involves an ether payment in line 10. So the metadata cloud be formulated as $S_w := \langle\phi, P_w\rangle$, where P_w only contain a payment $p_w : \langle g^*, \texttt{TokenBox}, I_s, \texttt{amount}, C\rangle$. And the condition of payment $C : \{c_0 \cdots, owner == I_s, \cdots c_k\}$ restricts that `msg.sender` (I_s) must be the `owner`.

E. Core Detector

The detection of UP is based on the collection of transaction metadata $S = \{S_1, S_2, \cdots, S_n\}$ generated during the process of symbolic execution. According to the definition in §III-A, SAFEPAY detects potential UP issues via searching for violations on the FVE oracle. More specifically, the detection works in a two-phase manner. In the first phase, SAFEPAY automatically selects all payments and generates a set of transaction sequences T, with a specific bound k. In the second phase, for each transaction sequence $T_i = tx_1, tx_2, \ldots, tx_n$ in T, SAFEPAY checks whether T_i holds a UP. Lastly, the *Core Detector* generates the UP information in a JSON report.

Algorithm 1: Detection of UP issues

Input : $S = \{S_1, S_2, \cdots, S_n\}$ is a set of transaction metadata. k is the bound of transaction scenarios.
Output: $X = \langle P_F, T_F\rangle$ is the set of detected UP issues.

```
1  X ← ⟨φ, φ⟩
   // Phases 1: Genrate a set of transaction sequences
2  for S_t ← ⟨U_t, P_t⟩ ∈ S do
3      if P_t = φ then
4          continue
5      for p ∈ P_t do
6          T ← T ∪ genTX(p, S, k)
   // Phases 2: Detect UP in transaction sequences T
7  for T_i ∈ T do
8      if checkF(T_i, S_{T_i})! = φ then
9          T_F ← T_F ∪ T_i
10         P_F ← P_F ∪ checkF(T_i, S_{T_i})
11 return X
```

Algorithm 2 describes the workflow of `genTX`, *i.e.*, a process to generate candidate transaction sequences which may trigger UP. Firstly, we push the transaction of payment p (tx_p) in transaction sequences T_p as the start element (line 1-2). Secondly, given a bound k, we iteratively generate transaction sequences with no more than k transactions (line 3-7) to avoid transaction space exploration. Specifically, for generating a new set of transaction sequences with m transactions (T_m), we traverse the first element tx_1 in transaction sequences T_{m-1} and insert its related transactions tx_{S_i} in front of the transaction sequences (line 5-7). The tx_{S_i} is selected by a function `checkI` which is a boolean function utilizing SMT solver to check whether the given data update set U_i in S_i affect the condition or state of tx_1 (line 6).

Algorithm 2: The `genTX` procedure

Input : $p = \langle g, s, r, v, C\rangle$ is a payment operation at x.
$S = \{S_1, S_2, \cdots, S_n\}$ is a set of transaction metadata.
k is a specific bound to limit the number of transaction.
Output: T_p a set of transaction scenarios for the payment p.

1 $T_p \leftarrow \{\{tx_p\}\}$
2 $T_1 \leftarrow T_p$
3 **for** m *in* `range`$(2, k)$ **do**
4 **for** $S_i \leftarrow \langle U_i, P_i\rangle \in S$ **do**
5 **for** $t_p = \{tx_1, tx_2 \ldots, tx_{m-1}\} \in T_{m-1}$ **do**
6 **if** `checkI`(U_i, tx_1) **then**
7 $T_m \leftarrow T_m \cup \{tx_{S_i}, tx_1, tx_2, \ldots, tx_{m-1}\}$
8 $T_p \leftarrow T_p \cup T_m$
9 **return** V

Algorithm 3: The `checkF` procedure

Input : $T = \{tx_1, tx_2, \cdots, tx_n\}$ is the transaction scenario with at least one payment.
$S_T = \{S_1, S_2, \cdots, S_n\}$ the set of transaction metadata.

Output: P is the UP $\langle \cdots p_t, p_n, \cdots\rangle$.

1 $P \leftarrow \phi$
2 $\mathcal{V}[\rightarrow_e] \leftarrow 0$
3 $\mathcal{V}[\rightarrow_t] \leftarrow \phi$
4 **for** $\langle p_t = \langle g_t, r_t, v_t, C_t\rangle, S_t = \langle U_t, P_t\rangle\rangle \in S_n$ **do**
5 **for** $s \in S_n$ **do**
6 **if** $s = p_t$ **then**
7 **break**
8 **if** $s \in U_t$ & $r_t \rightarrowtail s$ **then**
9 **if** $r_t \rightarrowtail l_s$ || (`checkR`(r_s, S_T) & $r_t \rightarrowtail m_s$) **then**
10 $\mathcal{V}[\rightarrow_t] \cup \{s\}$
11 **if** $s = \langle g_s, r_s, v_s, C_s\rangle \in P_t$ & $r_t = r_s$ **then**
12 $\mathcal{V}[\rightarrow_e] = \mathcal{V}[\rightarrow_e] + v_s$
13 **if** $\mathcal{V}[\rightarrow_e] > I_v$ & $\mathcal{V}[\rightarrow_t] = \phi$ **then**
14 **if** `checkR`(r_t, S_T) **then**
15 $P \leftarrow P \cup p_t$
16 **return** P

Lastly, Algorithm 3 describes the `checkF` procedure for detecting violations on FVE. Specifically, for a set of transaction sequences T generated in previous procedures and their metadata S_T, we analyze the value exchange details. To this end, all the operations in metadata are explored one by one (line 4-15). For a token update operation as defined in Definition 2, we capture the operation that affects the recipient r_t value of p_t (line 8-10). Specifically, if the operation is a update on local storage $u_l : \langle v, l, \{c_0, \cdots, c_k\}$ and the storage location l related to r_t, we add it to $\mathcal{V}[\rightarrow_t]$ (line 9). Otherwise, if the operation is a update on external storage via message call $u_e : \langle r_s, m_s, \{c_0, \cdots, c_k\}\rangle$, two conditions should be checked (line 9): The first condition, check whether the address r_s can be manipulated via a boolean function `checkR`. And `checkR` utilizes the SMT solver to check whether the address r_s is dependent on taint sources $\{I_s, I_d, H\}$ in the given transaction sequence. The second condition, check whether the recipient r_t is a part of input data m. If both conditions satisfied, this operation is regarded as a token transfer to the recipient r_t , and we push it into $\mathcal{V}[\rightarrow_t]$ (line 9-10). Moreover, for ether transfer operations, we check whether their receivers are the recipient of p_t. If so, the amount of ether attached to the transfer is accumulated (line 11-12). Based on FVE, we firstly check if the amount of ether to transfer out of the contract is larger than the amount of ether to be transferred in and there is no token transfer (line 13), then we check whether it can be manipulated by arbitrary address using the function `checkR` (line 14). The payment p_t that passes the above condition will be added to the UP set P (line 15).

IV. Evaluation

Implementation. We have instantiated SafePay as a detector for Ethereum unfair payments. Specifically, we used the `py-solc` [13] to compile smart contracts with different versions. Z3 [14] 4.6.0 was used as the SMT solver to check the feasibility of potential UP issues. In principle, SafePay is general and not specific to the current implementation. That said, any EVM based symbolic executor and SMT solver would fit in this setting.

Experiment setup. Our experiments were performed on a Linux machine with i9-9900K CPU@3.60GHz and 64GB of RAM with 16 MB SmartCache. For Z3, we set the timeout to be one second per request. The global timeout for the symbolic execution was set to 300 seconds for each contract.

Dataset. Two types of datasets were used in our evaluation.

Dataset-B contains both buggy and fixed versions of smart contracts related to payment attacks from two security benchmarks [15], [16].

Dataset-L includes a large set of real-world smart contracts from Etherscan [17]. 46,327 were used in total.

A. Comparison with Existing Analyzers

We evaluated the effectiveness of SafePay via comparisons with state-of-art analyzers on two types of datasets as aforementioned. The comparison tools included Oyente (v0.2.7) [3], Mythril (v0.21.15) [18] and Securify (2019-05-01) [4]. Since existing analyzers provided no direct support

TABLE II: UP detection result on **Dataset-B**. According to the benchmark contract, ✓represents a true positive or true negative and ✗ represents a false positive or false negative. $*$ indicates a bug-free contract.

Contract	Buggy Line	SAFEPAY	Securify	Mythril	Oyente
simple_dao.sol	17	✓	✓	✓	✓
DAO.sol	911	✓	✓	✗	✓
SpankChain_Payment.sol	422	✓	✗	✗	✗
modifier_reentrancy.sol	14	✗	✗	✓	✗
simple_dao_fixed.sol	$*$	✓	✓	✗	✓
modifier_reentrancy_fixed.sol	$*$	✓	✓	✗	✓

TABLE III: Detection results of unfair payment attacks on **Dataset-L**. For Securify and Mythril, two kinds of analysis were considered since they both targeted on payment-related issues.

Detector	Analysis	#Reported Attacks			
		#Unfair	#Fair	#Total	False Positive (%)
SAFEPAY	*Unfair Payment*	510	80	590	13.5%
Securify	*DAO in contract*	376	6,148	6,524	94.2%
	UnrestrictedEtherFlow in contract	33	584	617	94.6%
Mythril	*External call to user-supplied address*	53	1,099	1,152	95.3%
	Unprotected Ether Withdrawal	141	64	205	31.2%
Oyente	*Reentrancy Vulnerability*	31	87	118	73.7%

for UP, we used their results of payment-related analysis instead. Specifically, in the case of Oyente, we counted the number of *Reentrancy Vulnerability*. For Mythril, *External call to user-supplied address* and *Unprotected Ether Withdrawal* were considered. In terms of Securify, *DAO in contract* and *UnrestrictedEtherFlow in contract* were included, respectively. The comparison was designed not only to validate whether UP was able to cover more practical payment attacks than other heuristics, but to help develop in-depth understanding on blockchain payment security threats as well.

The results on **Dataset-B** is shown in Table II. For the four vulnerable contracts, namely simple_dao.sol, DAO.sol, SpankChain.sol and modifier_reentrancy.sol, SAFEPAY managed to detect three of them while other analyzers missed two or three potential attacks. Particularly, the SpankChain.sol contract which suffered from an attack with $38,000 loss escaped from all the detection except for SAFEPAY. In the case of two fixed versions of vulnerable contracts, *i.e.*, simple_dao.sol and modifier_reentrancy.sol, SAFEPAY generated no false alarms as Securify and Oyente. Unfortunately, Mythril considered both contracts as vulnerable. In sum, SAFEPAY produced the least number of false negatives and positives in this small-sized dataset among four analyzers considered.

Furthermore, the empirical results of a large-scale evaluation on **Dataset-L** is shown in Table III. For attacks reported by SAFEPAY, we manually validated them by checking whether FVE was satisfied in smart contracts. Then, we counted the number of unfair contracts (true positives) and fair ones (false positives), respectively. As shown in Table III, SAFEPAY reported 590 issues in the dataset, which was four times more than Oyente. However, this number was relatively small compared to both Securify and Mythril which reported 7,141 and 1,357 potential payment attacks in total. Although these two analyzers produced a big report, many cases from their reports were identified as false alarms. While the ratios of false positives for other three analyzers were over 70% (The analysis of *Unprotected Ether Withdrawal* in Mythril generated a comparatively small number of false positives.), the number for SAFEPAY was as low as 13.5%. That said, SAFEPAY is more semantically accurate *w.r.t.* unfair payments on blockchain. More specifically, instead of looking for certain code patterns on smart contract payments, the analysis in SAFEPAY tracks payment dependencies in a given contract and further determines whether fairness is guaranteed in all possible transaction scenarios. This way, SAFEPAY is able to avoid early reports on potential unfair payments which later turn out to be actually fair. On the other hand, there were a group of cases where the detection required analyzing multiple payments rather than a single one. Our evaluation observed misses on this type of attacks for existing analyzers. We will explain both situations in §V. All the evaluation results will be released six months after publication for security reasons.

B. Validation

To validate the UP problems reported by SAFEPAY, we set up reproducible attacks on Ethereum network and sent attacking transactions to vulnerable contracts. Specifically, we used the Metamask extension[†] to connect the browser with Ethereum. Then, we deployed contracts and sent transactions via Remix[‡]. For ethical reasons, all the validations were performed on the Ropsten testnet. Next, we describe the process of a specific contract.

[†]https://metamask.io/
[‡]http://remix.ethereum.org

```
1   contract Congress {
2    Proposal[] public proposals;
3    mapping (address => uint) public memberId;
4    address owner;
5    ①function ReconOwned() public {
6     owner = msg.sender;
7    }
8    modifier onlyOwner {
9     require(msg.sender==Owner);
10    _;
11   }
12  ②function addMember(address target) onlyOwner {...}
13  ③function newProposal(...) onlyMembers returns (
        ProposalID) {
14     proposals.push(...)
15   }
16  ④function executeProposal(uint p) public {
17     assert(proposals[p].call.value(p.amount)();
18   }
19  }
```

Fig. 5: A vulnerable smart contract with UP

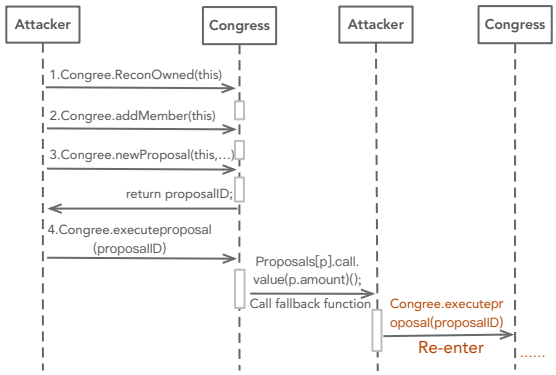

Fig. 6: Attack transaction scenario of Figure 5

The contract in Figure 5 manifests an unfair payment at line 17 since `proposals[p]` is externally controllable. We simulated a UP attack on the Ropsten testnet as demonstrated in Figure 6. The attack transaction sequences are: (1) An attacker calls `ReconOwned` ① updates `owner` to him/herself. (2) An attacker invokes `addMember`② to include him/herself in the collection of members. This can work because the attacker is considered as an owner. (3) An attacker calls `newProposal`③ to add him/herself into the proposals and get a `proposalID` from `Congress`. (4) An attacker uses the `proposalID` to execute `executeProposal`④ and iteratively send ether to him/herself via the payment at line 17. The validation transaction has been confirmed on Ropsten at `0xcb3198c3dc8f6fb0487aba5ca8fca6364e0db9bcbd004cb632f1eed1cb705faf`.

V. Discussion

In this section, we present an explanation on representative cases and discuss on false positives/negatives of SafePay.

TABLE IV: Representative cases. r_e and r_t denote recipients of ether and token payment, respectively.

Type	Contract
Fixed r_e	*Reservation*
Arbitrary r_e, Fixed r_t	*AkilosToken*
Arbitrary r_e and r_t	*IKyberNetworkProxy* *SpankChain_Payment* *ShortOrder*

A. Representative Cases

We highlight three representative types of contracts which we believe are important for designing an unfair payment analyzer, as in Table IV. Particularly, we use r_e and r_t to denote recipients of ether and token payment, respectively.

```
1  // 0xaaa6c933006e4d4b9a305015a69eed3d177fd355
2  contract Investment{
3    Crowdsale public ico = Crowdsale(0x0807....95b72a);
4    function buyTokens(uint _from, uint _to) {
5      ico.invest.value(amount)();
6      delete balanceOf[investors[i]]; }
```

Fig. 7: *Fixed* r_e payment.

Fixed r_e**.** Consider the *ether* payment at line 5 of Figure 7, the function `buyTokens` allows an ether payment to a fixed address `ico`. Since the recipient is set at the time of contract creation, the payment is a fair one. Existing tools could be misled in this case due to the state update (line 6) following the payment.

```
1    //0x1da73fc09ea07781482994036a0eecc7e6952dfb
2    contract TydoIco {
3      constructor(address _coinToken, ....){
4        token = _coinToken; }
5      function refund(ISetToken set, ...) public {
6        msg.sender.transfer(weiAmount);
7        token.transfer(owner, balances[msg.sender]);
8         ....  }
```

Fig. 8: *Arbitrary* r_e *and fixed* r_t pattern.

Arbitrary r_e***, Fixed*** r_t**.** Figure 8 shows another fair payment pattern in function `refund`. Specifically, two payments are created at line 6 and 7. While the payment at line 6 transfers ether to a given address, the payment at line 7 sends token to a predefined recipient `token`. Based on FVE, the two payments are fair in SafePay. However, existing analyzers report line 6 as a payment attack since the recipient `msg.sender` is dependent on transaction senders, leading to a false alarm.

Arbitrary r_e ***and*** r_t**.** Another representative case is a contract with multiple ether and token payments whose recipients are both arbitrary, as shown in Figure 9. Specifically, line 6 transfers *ether* to `msg.sender`, then a token payment is performed to `token` at line 8. In this case, a UP is triggered

```
1   //0xf91546835f756da0c10cfa0cda95b15577b84aa7
2   contract SetBuyer {
3
4     function buy(ISetToken set, ...) public {
5       if (address(this).balance > 0) {
6         msg.sender.transfer(address(this).balance);
7       }
8       require(token.transfer(msg.sender, ...); }
```

Fig. 9: *Arbitrary r_e and arbitrary r_t* pattern.

when the `token` blocks the token payment and repeatedly steals ether from the contract. In our evaluation, state-of-art analyzers considered the ether payment at line 6 as a potential attack because it pointed to an arbitrary recipient `msg.sender`. For the token payment at line 8, existing analyzers ignored it since the payments is associated with no ether and state updates followed. From the view of SAFEPAY, the UP in this contract is not equivalent to an unrestricted ether flow at line 6. The reason is the payment at line 6 is vulnerable only when the threat at line 8 is considered at the same time. In another word, given a secure token payment at line 8, line 6 becomes safe accordingly. Therefore, existing analyzers missed this unfair payment in the evaluation.

B. False Positives and Negatives of SAFEPAY

SAFEPAY also has some limitations. While these limitations are not only challenging for SAFEPAY, but for all state-of-the-art analysis tools. As shown in Table V, we summarized several classes of common patterns that we encountered. Such knowledge can greatly help the future work of researchers in this area.

TABLE V: Smart contracts expose SAFEPAY limitations

Type	Contract
Hash-based Sanity Check	*CompanyFundsWallet*
Gambling Contract	*casinoRoyale*
Honeypot Contract	*PinCodeMoneyStorage*
Misunderstanding on Payment	*AkilosIco*

```
1  // 0x6e6f819299e7809ce744f37fae9f84fe38d95f1c
2  contract CompanyFundsWallet{
3   bytes32 keyHash;
4
5   function withdraw(string key) public payable {
6    if(keyHash == keccak256(abi.encodePacked(key))) {
7      msg.sender.transfer(address(this).balance); }
```

Fig. 10: Hash-based Sanity Check

Pattern 1: Hash-based Sanity Check. Some contracts use a hash function to check the authority of the caller, which cannot be captured by SAFEPAY and cause some false positives. Figure 10 shows a sample contract of this pattern. In function `withdraw`, it determines whether the user is an authorized user by checking whether the hash of the `key` is equal to the storage `keyHash`. It is difficult to identify such specific conditions for constraining the caller from a set of path conditions.

```
1  // 0x919b5284182c676d02a3d657379c4f6e9e65eefd
2  contract casinoRoyale {
3
4   function flipCoin() public payable {
5    require(msg.value > 1500);
6    uint value = RandomGen.random(10,uint8(msg.value));
7    if (value > 55) {
8      msg.sender.transfer(msg.value * 2); }
```

Fig. 11: Gambling Contract

Pattern 2: Gambling Contract. There are some special smart contracts, *e.g.* gambling contracts, their functions naturally do not satisfy the FVE. Figure 11 shows a famous gambling contract named `casinoRoyale`. The player can invoke `flipcoin` to play the game. The `msg.value` is used to generate the random number `value` in line 6. If the check of `value` in line 7 passes, the player will receive a double value reward in line 8. The only way to avoid false positives caused by such contracts is through analyzing the fairness in conjunction with the contract function.

```
1  // 0x9efc7a38552e63534a8e9b9558adabd73297f91d
2  contract PinCodeMoneyStorage {
3   address private Owner = msg.sender;
4   uint public SecretNumber = 95;
5
6   function Withdraw() public payable{
7     require(msg.sender == Owner);
8     Owner.transfer(this.balance); }
9   function Guess(uint n) public payable {
10    if(msg.value >= this.balance) {
11      if(n*n/2+7 == SecretNumber ) {
12        msg.sender.transfer(this.balance+msg.value);
```

Fig. 12: Honeypot Contract

Pattern 3: Honeypot Contract. During our empirical analysis, we noticed a few cases flagged by SAFEPAY are honeypots, which means attackers might lose the ether if they exploit them. Figure 12 shows an example contract named `PinCodeMoneyStorage`. The attacker can easily calculate the `n` that satisfies the condition in line 11 and then get all ether by the payment in line 12, which clearly does not satisfied FVE. But the contract takes advantage of the fact that `this.balance` could be easily modified, such as invoking `withdraw`. As a result, the condition in line 10 will always fail and the transfer will never occur. Since the traps of smart contract honeypots vary in sophistication, it is hard to define heuristics to expose them from UP.

Pattern 4: Misunderstanding on Payment. Due to the specific identification of token payment, some UP are missed by SAFEPAY. Figure 13 shows a sample contract which holds a missed UP. Line 6 performs an *ether* payment to an arbitrary caller, and the following external call in line 7 will be considered as a token transfer since the `token` is a fixed

```
1  // 0x0b1e9e95f0655716ee00ae455caf9ba01364491a
2  contract AkilosIco{
3   token = new AkilosToken();
4   function participate(address participant,uint value){
5      uint256 tokenCount = safeMul(value, exchangeRate);
6      msg.sender.transfer(value);
7      token.mint(msg.sender, tokenCount); }
8
9  contract AkilosToken{
10  function mint(address _to, uint value) onlyMinter{
11     balances[_to] = safeAdd(balances[_to], _value); }}
```

Fig. 13: Misunderstanding on Payment

address and the recipient of payment in line 6 is a part of input data. When we check the code of `mint` (line 10-11), we will find that the function will increase token of `msg.sender` instead of reducing. Hence, the payment in line 6 is a UP.

VI. Related Works

In recent years, the security of smart contracts has been a hot research topic. There has been a lot of research regarding smart contract vulnerabilities and common security models [19], [20]. To audit smart contracts, researchers developed statistic analyzers using symbolic execution [3], [4], [7], [18], [21], [22]. Oyente, Mythril focus on finding an execution path that satisfies a given property, which utilizes Z3 SMT solver [14] to decide the satisfiability [3], [18]. Securify is based on abstract interpretation and encode the smart contract into Datalog formulas [4]. In a dynamic analysis of smart contract, ContractFuzzer is proposed a novel fuzzer to analyze smart contracts with testing oracles defined for several known vulnerabilities [5], [23]. MuSC explored the potential of applying mutation testing into smart contracts [24].

VII. Conclusion

In this paper, we highlighted a new security oracle called UP for Ethereum smart contracts and a systematic framework SafePay to detect UP issues as well. Specifically, the oracle is more general than existing ones in terms of modeling attack scenarios. Our approach leverages symbolic execution to generate both transaction dataflow and metadata of a given contract. Furthermore, SafePay infers potential UP via checking the satisfiability of oracle violations. We have instantiated SafePay and applied it in analyzing real-world Ethereum contracts. SafePay managed to identify more vulnerable contracts with real exploits than analyzers in the literature. Extensions of SafePay for more security problems are considered as future work.

VIII. Acknowledgement

Han Liu is the corresponding author. Lei Wang is supported by National Key Research and Development Program of China (No.2019YFB2101601).

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.
[2] Ethereum, "Solidity — solidity 0.4.19 documentation," 2017. [Online]. Available: https://solidity.readthedocs.io/en/develop/
[3] 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.
[4] 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.
[5] 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.
[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] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." NDSS, 2018.
[8] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu, "Kevm: A complete semantics of the ethereum virtual machine," Tech. Rep., 2017.
[9] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 520–535.
[10] A. Bahga and V. K. Madisetti, "Blockchain platform for industrial internet of things," *Journal of Software Engineering and Applications*, vol. 9, no. 10, p. 533, 2016.
[11] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *arXiv preprint arXiv:1802.06038*, 2018.
[12] 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.
[13] Ethereum, "py-solc," https://github.com/ethereum/py-solc.
[14] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
[15] Mythril, "Smart contract weakness classification registry," https://github.com/SmartContractSecurity/SWC-registry.
[16] trailofbits, "Smart contract weakness classification registry," https://github.com/crytic/not-so-smart-contracts.
[17] E. Team, "Etherscan: The ethereum block explorer," 2017.
[18] C. Diligence, "Mythril(2018)," https://github.com/ConsenSys/mythril-classic.
[19] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 79–94.
[20] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 494–509.
[21] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 814–819.
[22] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun, "Enabling clone detection for ethereum via smart contract birthmarks," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 105–115.
[23] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 259–269.
[24] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen, "Musc: A tool for mutation testing of ethereum smart contract," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1198–1201.