An Ever-evolving Game: Evaluation of Real-world Attacks and Defenses in Ethereum Ecosystem

Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao†, Min Yang, and Yuan Zhang
Fudan University, † Johns Hopkins University
{sfzhou17, yangzhemin, jxiang17, m_yang, yuanxzhang}@fudan.edu.cn, † yinzhi.cao@jhu.edu

Abstract

Smart contract security has drawn much attention due to many severe incidents with huge ether and token losses. As a consequence, researchers have proposed to detect smart contract vulnerabilities via code analysis. However, code analysis only shows what contracts can be attacked, but not what have been attacked, and more importantly, what attacks have been prevented in the real world.

In this paper, we present the first comprehensive measurement study to analyze real-world attacks and defenses adopted in the wild based on the transaction logs produced by uninstrumented Ethereum Virtual Machine (EVM). Specifically, our study decouples two important factors of an adversarial transaction—i.e., (i) an adversarial action exploiting the vulnerable contract and (ii) an adversarial consequence like ether or token transfers resulted from the action—for the analysis of attacks and defenses.

The results of our study reveal a huge volume of attacks beyond what have been studied in the literature, e.g., those targeting new vulnerability types like airdrop hunting and those targeting zero-day variants of known vulnerabilities. Besides successful attacks, our study also shows attempted attacks that are prevented due to the deployments of defenses. As the nature of cyber-security, those defenses have also been evaded, mainly due to incomplete defense deployments. To summarize it, we believe that this is an ever-evolving game between adversaries obtaining illegal profits and defenders shielding their own contracts.

1 Introduction

Smart contract security has drawn much attention as the emergence of several famous, multi-million-dollar incidents, such as TheDAO attack [42] and the Parity Wallet Hack [34], which steal thousands of ethers and tokens from the Ethereum ecosystem. One lesson that we have learned from those incidents is that smart contracts, just like normal computer programs, have vulnerabilities—such as integer overflow, reentrancy [4], and call injection (or called code injection [30])—and even honeypot [37, 38].

In the past, researchers propose using code analysis [5, 9, 23, 27–33, 35, 36, 39–41, 43], e.g., static and dynamic, to detect those vulnerable contracts and honeypots. On one hand, many static analysis tools adopt either source- or bytecode-level analysis to find vulnerabilities based on certain code patterns. However, those static analysis tools can only determine whether a contract is vulnerable but not whether or how it is exploited in the real world. For example, a recent report from Perez and Livshits [35] has already shown that only a small number, i.e., around 2%, of vulnerabilities found by six recent prior works [27, 29–31, 33, 41] are actually exploited.

On the other hand, some dynamic analysis tools [35, 36] are proposed to detect and understand, to some extent, what attacks have been adopted in the real world. For example, Sereum [36], a dynamic analysis tool of reentrancy attacks, analyzes the first 4.5 million transactions on Ethereum blockchain and finds several unknown reentrancy attack patterns and vulnerable contracts. The aforementioned report from Perez and Livshits also modified Ethereum Virtual Machine (EVM) to perform dynamic analysis and understand whether a reported contract has been exploited. However, dynamic analysis tools, especially those which propagate taints, are usually heavyweight and not scalable to a large-scale measurement.

The research task that we are tackling in the paper is to analyze all the existing transactions on the Ethereum blockchain and understand what strategies adversaries have adopted in real-world and how prevalent and successful those strategies are. This task is beyond what prior code analysis, either static or dynamic, can handle: We aim to analyze both prior attacks—no matter succeeded or failed—and defenses using public information that has already been outputted by EVM during the execution of transactions.

Particularly, in this paper, we perform the first comprehensive study of 420 million Ethereum transactions from August 2015 to March 2019 and measure real-world adoptions of attacks and defenses. Our methodology, at its core, is a transaction log analysis that matches execution traces outputted by uninstrumented EVM against so-called adversarial

transaction signatures and looks for adversarial transactions, either confirmed (i.e., successful) or attempted (i.e., failed). Our signature matching involves two steps, which decouples two important concepts in adversarial transactions, i.e., (i) an adversarial action and (ii) an adversarial consequence. The former, like a function call with certain parameters, shows the intent of the transaction to exploit a contract, and the latter, such as an ether transfer, shows the result of the former in exploiting the contract.

Here are the two steps in details. First, we design a so-called action clause of the adversarial signature to match contract interactions in the transaction log and to decide whether a transaction has an adversarial intent in exploiting a vulnerability. Particularly, we construct a special structure, called *action tree*, for each transaction or contract, which represents all the inter-contract interactions, such as function calls, contract creation and contract destruction. Then, we match the action clause against those action trees to find adversarial transactions.

Second, we design another clause of the signature, called result clause, to match ether, token, or ownership transfers between contracts in the log and confirm the consequences of adversarial transactions. Particularly, we build another data structure, called *result graph*, to represent all such transfers between contracts for each transaction or contract. Then, we match the result clause against the constructed result graphs to confirm the consequences, thus finding confirmed adversarial transactions.

One major outcome of our study is to reveal what attack strategies have been adopted in practice and what consequences of these attacks are. We have observed a clear gap between what prior works have found and what attackers adopt in the real world. Particularly, 93.55% of confirmed adversarial transactions are targeting 198 vulnerable contracts using a new attack tactic, i.e., airdrop hunting. We have also observed a big shift of attack strategies over time. In the early days of Ethereum, i.e., from August 2015 to August 2017, reentrancy and call injection dominates all the adversarial transactions, taking up 97.00% of all the confirmed. Then, the attack focus gradually shifts to integer overflow and airdrop hunting: From September 2017 to March 2019, 76.05% of attempted and 98.12% of confirmed adversarial transactions are caused by these two attack categories.

Another outcome of our study is to reveal real-world, deployed defenses. Particularly, we analyzed those attempted but not confirmed adversarial transactions and then their target contracts to find adopted defense strategies. In total, we find six classes of defenses adopted by 5.8 million open-source contracts. There are two major widely-deployed defenses: *SafeMath* adopted by 3.1 million contracts for arithmetic operations and the *onlyOwner* check by 2.1 million. These deployed defenses are indeed effective in defending against 1,276 attempted adversarial transactions: The *SafeMath* is the most effective one that prevents 1,161 adversarial transactions.

Some of those defenses, although deployed, are also being evaded mostly due to incorrect or inappropriate deployments. In total, we have observed 68,873 adversarial transactions that have successfully evaded defenses deployed by existing contracts. For example, one Ethereum Request for Comment 20 (ERC20) token contract suffers a successful integer overflow attack because it uses *SafeMath* functions in all the ERC20 interfaces but not a customized token transfer function. We believe that the attack and defense in the Ethereum ecosystem will be an ever-evolving game between two parties.

Apart from existing attacks and defenses, one byproduct of our study is the detection of zero-day vulnerable contracts. Particularly, once we identified a transaction as confirmed adversarial, the target contract is obviously vulnerable. Further, if the contract is firstly considered by our study as vulnerable, we can treat the vulnerability as zero-day. The main reason for the discovery of zero-day vulnerabilities is the imprecision of existing code analysis, while a log analysis used in our study is in parallel to prior code analysis. For example, some prior works cannot perform cross-contract analysis [5, 41]; some have coverage issues that skip sensitive multi-target token transfer functionality [29]; some only perform dataflow analysis on basic data type but not complex ones like objects [5, 9].

We find 22 zero-day vulnerabilities, e.g., integer overflow and reentrancy, and 51 zero-day honeypots with real-world adversarial transactions. Those zero-day vulnerabilities are indeed exploited in the real world and somewhat popular. Take integer overflow for example. 39.93% of all the confirmed adversarial transactions targeting integer overflows belong to 16 previously-unknown vulnerabilities found by our study.

Lastly, in the spirit of open science, we have made our measurement study results available in this URL (https://drive.google.com/open?id=1xLssDxYWyKFCwS5HUrQaSex0uwJRSvDi). We have also reported all the zero-day vulnerabilities to their developers—if the contracts are open-source and developers are available—and also CVE database.

2 Overview

In this section, we start from a running example to explain our methodology and then describe our threat model, i.e., in-scope and out-of-scope attacks.

2.1 A Running Example

In this subsection, we illustrate a concrete attack example—namely airdrop hunting—to describe our methodology in detecting and modeling real-world attacks and defenses. Particularly, airdrop is a crypto-token feature that distributes new participants a fixed, small amount of tokens as a way of gaining attention and attracting followers. Airdrop hunting is a relatively-new attack strategy that exploits the weaknesses of airdrop and bypasses the identity check of new participants to obtain a large number of free tokens.

```
1  contract Simoleon is ERC20Interface {
2      function transfer(address _to, uint256
           _amount) returns (bool success) {
3          initialize(msg.sender);
4          ...
5      }
6      function initialize(address _address)
           internal returns (bool success) {
7          if (!initialized[_address]) {
8              initialized[_address] = true;
9              balance[_address]=_airdropAmount;
10         }
11     }
12 }
```

Figure 1: A vulnerable airdrop contract example.

```
1  modifier isHuman() {
2      address _addr = msg.sender;
3      uint256 _codeLength;
4
5      assembly {_codeLength := extcodesize(_addr
           )}
6      require(_codeLength == 0, "humans only");
7      _;
8  }
9  modifier anotherIsHuman() {
10     require(tx.origin == msg.sender, "humans
           only");
11     _;
12 }
```

Figure 2: An airdrop hunting defense example.

Figure 1 shows a vulnerable, real-world, ERC20 token contract, called *Simoleon*—the contract only checks the identity of a participant based on its `msg.sender` (Line 3) and then distributes airdrops if the `msg.sender` is new and never seen before. However, a new `msg.sender` may belong to a contract generated automatically by an adversary hunting for airdrops. Specifically, we show the execution traces of an airdrop hunting transaction in Table 1, in which the master contract controlled by the adversary creates 50 slaves to hunt airdrops via calling the `transfer` function. All the slaves transfer the airdrops to the master contract and then destroy themselves to avoid being directly tracked.

Now, we use this running example to explain our measurement study. From a high-level, our study has three sub-analysis: (i) attack analysis, i.e., finding adversarial transactions, (ii) defense analysis, i.e., finding contracts and corresponding defenses with attempted adversarial transactions, and (iii) evasion analysis, i.e., finding adversarial transactions evading existing defenses. We describe those three respectively using the example.

First, the attack analysis finds adversarial transactions like those with execution traces as shown in Table 1. These adversarial transaction traces have two patterns, an adversarial action that exploits the vulnerable contract and an adversarial consequence showing that the adversary illegitimately obtains tokens. Specifically, the action here, for an airdrop hunting attack, is that the master contract creates many slave contracts, which then call a token transfer function in the victim contract. Subsequently, the consequence here is that slaves collect airdrop bonus and then transfer them back to the master.

Second, the defense analysis starts from attempted adversarial transactions like those that are similar to traces in Table 1 but failed, and then finds corresponding defenses that lead to the failure of adversarial transactions. That is, although these transactions have adversarial actions, but do not have any adversarial consequence: tokens are not obtained by the slaves and then the master.

Here is one example defense, i.e., an `isHuman` modifier in Figure 2, against airdrop hunting. This modifier—found in a famous gambling contract Fomo3D [8] and used by 36 airdrop token contracts—checks the code length of a participant and decides whether it is a contract created by another contract or a human. Therefore, if an adversary generates many slave contracts automatically, the code length of each slave will be larger than zero, thus being blocked.

Lastly, the evasion analysis finds confirmed adversarial transactions that bypass defenses found in the previous analysis. The aforementioned `isHuman` modifier can be evaded with confirmed adversarial transactions because an adversary can embed the airdrop hunting code in the constructor function, in which the code length is zero when the victim contract measures the yet-to-be-constructed slaves. Of course, the defenders also fight back with another modifier, i.e., the `anotherIsHuman` in Figure 2. This defense checks the transaction initiator (`tx.origin`) and the airdrop participant (`msg.sender`) to ensure that the participant is not a slave invoked by a master.

2.2 Threat Model

Intuitively, in this study, we measure existing attacks with explicit, gaugeable losses in terms of ethers and tokens. For example, if an adversary's contract exploits a vulnerability of a victim contract and then gains say 100 ethers from the victim, we would consider this attack as in-scope. For another example, if an adversary makes a victim contract unavailable to others, e.g., via an out-of-gas attack [27] or lock of ether as in the famous Parity Wallet Freeze[1] [6], the adversary does not directly obtain any ethers or tokens and therefore we consider it as out-of-scope. We adopt this threat model because the attacks with explicit losses can be quantified and measured.

Formally, our threat model includes contract-level attacks that lead to an explicit ether or token flow or an ownership transfer from one contract, e.g., a victim, to another, e.g., the adversary. For example, a reentrancy attack will lead to a repeated transfer of ethers or tokens from the victim to the adversary, thus considered as in-scope. By contrast, the aforementioned denial-of-service and blockchain-level attacks like the replay attack [19] are out-of-scope.

In-scope Attacks Now, for the convenience of readers, we show a list of all the in-scope attacks considered in the paper

[1]Note that "Parity Wallet Freeze", due to a glitch in the multi-sig library, is different from another famous "Parity Wallet Hack" [34] caused by a call injection vulnerability.

Table 1: Example traces of an airdrop hunting transaction targeting the vulnerable contract in Figure 1. Each row, called a trace, shows an interaction between two contracts in the "From" and "To" columns. In particular, a trace includes certain amounts of ethers ("Value" column), binary data ("Payload" column) as payload, and whether the interaction succeeds ("Status" column). The "Address" column indicates how the trace is related to others of the transaction.

Address	From	To	Payload		Type	Value	Status
			Entry function	Parameters			
NULL	Attacker	Master	0x2b6cab44	0x32	call	0	Success
↘0	Master	$Slave_1$	N/A	N/A	create	0	Success
↘0,0	$Slave_1$	Victim	transfer(address,uint256)	_to: Master, _amount: 1,000,000	call	0	Success (or Failed)
↘0,1	$Slave_1$	Master	N/A	N/A	suicide	0	Success
...	...	...	...	...	...	...	
↘49	Master	$Slave_{50}$	N/A	N/A	create	0	Success
↘49,0	$Slave_{50}$	Victim	transfer(address,uint256)	_to: Master, _amount: 1,000,000	call	0	Success (or Failed)
↘49,1	$Slave_{50}$	Master	N/A	N/A	suicide	0	Success

below and explain them.

- *Airdrop hunting.* Airdrop hunting, as described in our running example (Section 2.1), leads to token flows from the victim contract to the master controlled by the adversary.
- *Call injection.* Call injection, which allows any contract, including adversaries, to call a sensitive function in a vulnerable contract, is often used to make an ownership change and initiate ether or token transfers.
- *Reentrancy.* Reentrancy, as mentioned, usually leads to repeated token or ether transfers from the victim to the adversary.
- *Integer overflow.* Only some integer overflow attacks target a variable recording the token amount owned by a victim, followed by an adversary transferring the overflowed amount. Those attacks are in-scope and other integer overflows like those causing a denial-of-service are not.
- *Honeypot.* A honeypot lures a victim to transfer some ethers or tokens and then participate with bait but no actual paybacks.
- *Call-after-destruct.* Call-after-destruct is the invocation of a function in a destructed contract with ethers, leading to the loss of these ethers forever. Noted that the call-after-destruct is different from an out-of-scope suicidal attack, in which an adversary exploits an unprotected interface and destroys the victim contract.

3 Methodology

In this section, we describe our measurement methodology.

3.1 Measurement Workflow

We now describe the overall workflow of our analysis as shown in Figure 3, which can be roughly divided into four phases. First, in phase (1), we perform several preparation works including (i) the construction of action trees and result graphs, i.e., two special representations, from execution traces and (ii) the manual generation of adversarial transaction signatures, containing both action and result clauses, for different vulnerability types. Second, in phase (2), we perform an attack analysis to detect both confirmed and attempted adversarial transactions using our adversarial transaction signatures. The action clause is matched against the action tree to find adversarial transactions, and then the result clause is matched against the result graph to confirm them. Third, in phase (3), we perform a defense analysis to understand why certain adversarial transaction fails. We extract the snippet of code that defends against adversarial transactions and find more contracts that adopt these defenses via code similarity analysis. Lastly, in phase (4), we look back at these confirmed adversarial transactions and analyze whether they can penetrate contracts with defense code via an evasion analysis.

3.2 Preparation Phase

In the preparation phase, we convert execution traces of transactions to special representations, i.e., action tree and result graph. At the same time, we generate adversarial transaction signatures to match with those two special representations in the attack analysis.

3.2.1 Action Tree and Result Graph

In this subsection, we discuss the construction of two important representations, i.e., action tree and result graph. The purpose of an action tree is to capture the actions that one contract performs upon another and represent them in a tree-like structure, and the purpose of a result graph is to capture the consequences of performed actions and represent them in a graph-like structure.

Definitions We now give the definitions of these two representations.

- *Action Tree.* An action tree is a representation of actions in an ordered tree-like structure, in which each node is a contract and each edge is an action from the source contract to the destination. An action, defined as what one contract performs upon another, has three concrete types: `create`, `suicide`, and `call`. `create` means that a contract creates a new contract in the destination address, `suicide` represents that a contract removes all its code and transfers all the ethers it owns to the destination contract, and `call` means that one contract calls another contract's function,

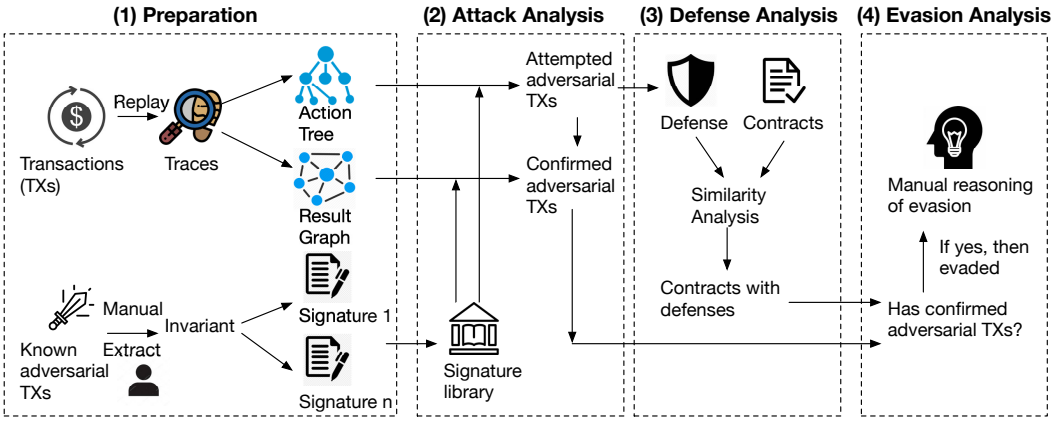


Figure 3: Measurement Workflow.

which could be an explicitly specified function or the default fallback function. In addition to its type, each action is annotated with three additional properties: the invoked function signature (or function definition if available), function parameters, and trace address (which is used to order edges).

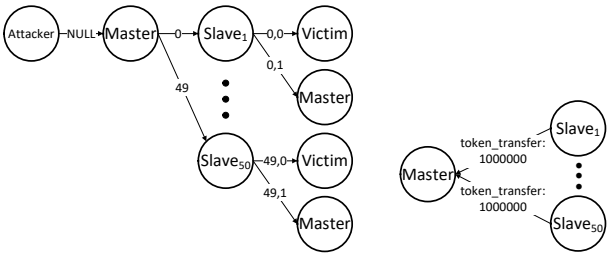
- *Result Graph.* A result graph is a representation of results in a graph-like structure, in which the nodes are unique contracts and the edges are sensitive results, i.e., ether transfer, token transfer and ownership change, which happens from one contract to another. Each edge in a result graph is annotated with the number of transferred ethers or tokens if applicable.

Note that these two representations have variations, i.e. either transaction- or contract-centric: Different variations can be used in the detection of different adversarial transactions. We now introduce them separately.

Transaction-centric Construction Transaction-centric construction is to convert the execution traces of each transaction into these two representations, i.e., action tree and result graph. First, we construct a transaction-centric action tree by following the initiating contract and the sequence of all the actions under that contract and creating edges from the initiator to the destinations. We then repeat the process until all the actions in the traces have been used in the construction.

Second, we construct a transaction-centric result graph by following all the actions and finding out their corresponding results for annotation. There are two sources, i.e., action raw traces and function parameters, to annotate the graph. (i) Ether transfer values are available in the raw trace associated with the action. (ii) Ownership and token transfer values are obtained from function parameters if the corresponding function signature matches the one documented by ERC standards as shown in Appendix B and the function call succeeds.

Now let us look at the construction of action tree and result graph (Figure 4) of our running example. We start from the first record in the traces, i.e., the row with the address NULL



(a) Action tree. (b) Result graph.

Figure 4: Two representations of the example traces in Table 1.

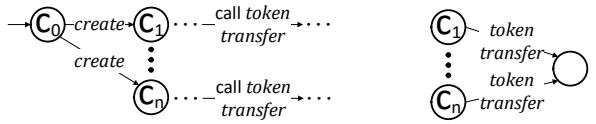
in Table 1, as the root node to construct action tree. The rows with the addresses from 0 to 49 are the direct children of the root node and then nodes with "0,0" and "0,1" are children of the "0" node. The fully constructed action tree is shown in Figure 4a. Next, we will extract the function parameter of each transfer call and construct a result graph annotated with transferred token values as shown in Figure 4b.

Contract-centric Construction Contract-centric construction is to convert the execution traces of all the transactions belonging to one contract to our special representations. Contract-centric representations are useful to capture the malice of contract-specific behaviors, such as honeypot. We construct contract-centric representations from transaction-centric ones. Here are the details. First, we locate all the transaction-centric action trees that contain the target contract and merge all these trees together in chronological order based on the target as the root node and other contracts that perform an action upon the root as the leaves. Second, we also merge all the result graphs that contain the target and construct a bigger result graph by merging duplicate nodes.

3.2.2 Adversarial Transaction Signature

In this subsection, we first describe our signature definition and then present how to generate signatures.

Definition An adversarial transaction signature has two clauses: action and result. The action clause of a signature



(a) Action clause. (b) Result clause.

Figure 5: Adversarial transaction signatures for airdrop hunting (a blank cycle represents contracts with no address constraints).

is a tree structure that starts from a node C_0 and provides the matching conditions of each level of the tree including edge properties and contract addresses. Figure 5a shows an example action clause of our airdrop hunting example: the C_0 is the starting node, which has properties like outgoing edges of *create*, and then the second-level nodes will eventually call token transfer function in some of the deeper levels. The three vertical dots in a level of the tree indicate that there could be more than one such similar node with a *create* as incoming edge, and the three horizontal dots across levels indicate that there could exist more than one node in between this token transfer edge and the previous node.

The result clause of a signature is a graph structure in which node names may be from those in the action clause and edges provide corresponding matching conditions. Figure 5b shows an example result clause of our airdrop hunting example. Nodes C_1 to C_n are from the second level of the action clause and they all have an outgoing edge to an arbitrary node.

Signature Generation In this part, we describe how to generate adversarial transaction signatures for attack analysis. Our generation has two steps: (i) invariant extraction, and (ii) human reasoning. In the first step, we extract common nodes and edges, called invariants, from action trees and result graphs of existing, known adversarial transactions. Then, in the second step, we rely on human experts to reason about the correctness of extracted invariants and add or remove constraints based on the attack semantics. Our signatures are opportunistic, and that we do not claim completeness because our purpose is to perform a measurement study of deployed attacks in the real world rather than detection of all the attacks. We will have an estimation of false positives and negatives of our measurement results in Section 4.

Now let us use our airdrop hunting example to describe the procedure of generating adversarial signatures shown in Figure 5. First, we extract common sub-trees and sub-graphs based on the representations of adversarial transactions similar to those in Figure 4. The common sub-tree is that a contract creates many contracts and newly-created ones call the token transfer of a victim and then destroy themselves. The common sub-graph is that newly-created contracts transfer a certain amount of tokens to another contract.

Second, we will manually examine the extracted sub-tree and sub-graph to generate both clauses of an adversarial transaction signature. The manually-collected airdrop hunting attacks typically create at least ten slaves, and we set the threshold of slaves in the sub-tree and sub-graph as two to detect all the slave creation transactions. Then, we delete the destroy action from the sub-tree because this is not a necessary step of airdrop hunting though performed in all the collected adversarial transactions. We also change the destination contract from the master to an arbitrary one as an adversary can transfer tokens to any contract.

3.2.3 Signature Library

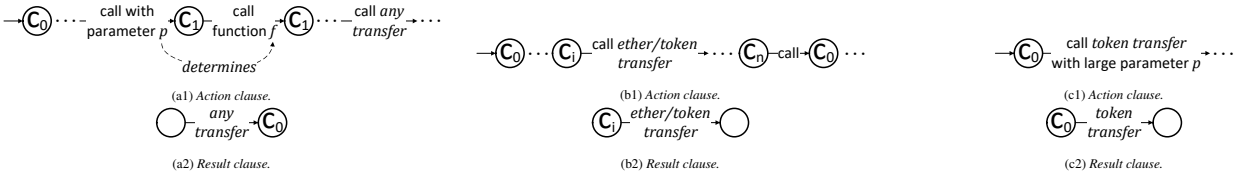
In this part, we list all our adversarial transaction signatures generated in our library based on the attack and signature type.

Transaction-centric Signatures We first describe three attack types that require only transaction-centric signatures in Figure 6.

- *Call injection.* The action clause (Figure 6a1) is that a contract calls its own function, which usually authorizes the contract itself, in an inter-contract way and the called function further proxies sensitive function calls, e.g., a transfer or ownership change, which is specified by a parameter from injected function call. The proxied function name can be embedded in a function parameter via two ways: (i) function signature and (ii) utf-8 encoded function name. Next, the result clause (Figure 6a2) specifies that the injected function call benefits any of the ancestor nodes, i.e., C_0, in the action tree in terms of ethers, tokens or ownership.
- *Reentrancy.* The action clause (Figure 6b1) is that a contract (C_0) calls another contract (C_i), which may call some other contracts but eventually will call C_0, and such looped invocation behavior will involve at least one transfer function. The result clause (Figure 6b2) is that the result edge caused by the transfer function in the loop of action tree may point to another contract of the adversary outside the loop.
- *Integer Overflow.* The action clause (Figure 6c1) is that a contract (C_0) calls a known sensitive token transfer function that contains a parameter, i.e., a value bigger than 10^{72} being close to the maximum range of signed 256-bit integer, to trigger the vulnerability. Next, the result clause is that C_0 transfers tokens to another contract belonging to an adversary.

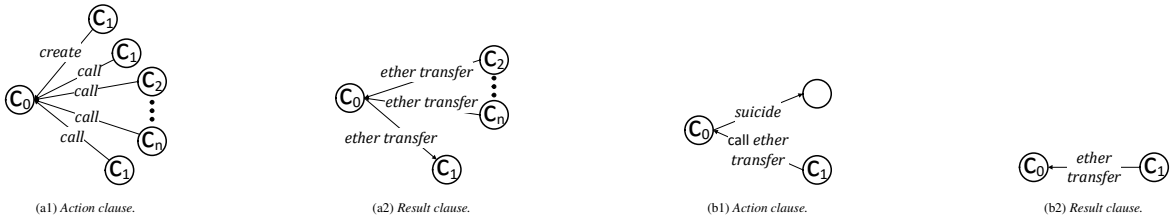
Contract-centric Signatures We now describe another three attack types that require contract-centric signatures in Figure 7.

- *Honeypot.* An action clause (Figure 7a1) is that the honeypot (C_0) is created and set up by another contract (C_1) and then accepts function calls from other non-owner contracts (e.g., C_2 to C_n). In the end, C_0 suicides and transfers collected ethers to C_1. A result clause (Figure 7a2) is that C_1, although first makes investment, benefits from C_0 and other contracts that transfer ethers to C_0 get no payback.



(a) Call injection. (b) Reentrancy. (c) Integer overflow.

Figure 6: Transaction-centric signatures.



(a) Honeypot. (b) Call-after-destruct.

Figure 7: Contract-centric signatures.

- *Call-after-destruct.* An action clause (Figure 7b1) is that C_0 first suicides and then another contract, e.g., C_1, still calls with ether transfer of C_0. A result clause (Figure 7b2) is that ethers are transferred from C_1 to C_0.

3.3 Attack Analysis Phase

Our attack analysis matches adversarial transaction signatures against action trees and result graphs of transaction execution traces. The analysis, by its nature, has two stages: action and result clause matching. The former finds attempted adversarial transactions, and the latter confirms those adversarial transactions.

3.3.1 Action Clause Matching

We match action clause by traversing through all the nodes in the action tree. The first step is to match the root node C_0 and if the root matches, we will match further levels. Then, if all levels match, we consider the action tree matches with the specific action clause. Let us again use the airdrop hunting example in Figure 1 to describe the matching. When we traverse through an action tree, we will find that the master node in Figure 4a matches with C_0 in Figure 5a because C_0 has many `create` actions on the outgoing edges. Then, C_1 to C_n also matches with $Slave_1$ to $Slave_{50}$, because they all call token transfer function. Since all the nodes and edges in the action clause are matched, we consider that the action tree in Figure 4a is a match, i.e., at least an attempted adversarial transaction.

3.3.2 Result Clause Matching

We perform result clause matching by checking each node and edge. Specifically, during action clause matching, we have recorded all the node addresses and matches them with real-world contracts. In this matching, we will confirm that the result graph also has corresponding nodes and edges. Here is how it works in our airdrop hunting example. Since we know that C_1 to C_{50} are slave contracts, we will see whether they have transferred all the tokens to another contract. In the case of Figure 4b, all the slaves transfer tokens to the master, which is C_0. That is, the result clause matches the result graph as well, which confirms the adversarial transaction.

3.4 Defense Analysis Phase

Our defense analysis phase has two steps: (i) behavior-based security check (i.e., the defense) identification, and (ii) extended defense mining with similarity analysis. Let us start from the first step. Our observation here is that most smart contracts implement defenses via Solidity functions that affect control flows [20], such as `require`, `assert` and `revert`, to abort an execution if being attacked. That is, if an attempted transaction fails to meet the conditions in these functions, its trace returns directly with a `Reverted` error. Therefore, we extract the control-flow-related statements that cause the failure as the security checks for the second step.

Second, we perform a backward dataflow analysis from the security check to extract all the sources of the check. Then, we use the security check and all the sources as the basis for the similarity analysis. The insight here is that once two contracts perform the same check on common input sources, they tend to use the same defense tactic. Therefore, we extract such a backward dataflow for all the open-source contracts and compare the extracted dataflow with the one with a certain defense. If both the security check and the sources match for these two contracts, we will consider that the target contract adopts the same defense.

We now look at a concrete example, i.e., the `isHuman` modifier in Figure 2. We first extract the security check that leads to a failed transaction, which is at Line 6. Then, we perform a backward dataflow analysis to find all the sources used in the check, in this case, the return value of `extcodesize()` at Line 5. Lastly, we find similar contracts by searching for

the use of `extcodesize()` and the comparison of the return value of `extcodesize()` with zero.

3.5 Evasion Analysis Phase

The purpose of our evasion analysis phase is to understand whether existing defenses have been evaded by new attacks. The analysis has two steps. First, we will analyze the contracts with defenses found in our defense analysis and see whether such contracts have confirmed adversarial transactions. Second, if these contracts have confirmed adversarial transactions, i.e., they are being penetrated regardless of the defense, we will further confirm and reason whether the adversarial transactions have indeed bypassed the corresponding security check adopted in the defense.

4 Implementation and Manual Analysis

In this section, we start from our implementation and preliminary results produced from automatic analysis. Then, we describe our manual efforts in reducing the false positives and estimating the false negatives.

4.1 Implementation and Preliminary Results

Our implementation of attack analysis is in 3,977 lines of Python code. We apply our implementation on execution traces from public service, particularly the Google BigQuery traces [12], for analysis. Note that the traces obtained from Google are the same as what we execute EVM in an archive mode ourselves and we adopt Google's traces to save execution time and storage space. The snapshot that we adopted has 1,063,473,983 rows of trace records of 420 million transactions until March 2019. The preliminary results of our study are shown in Table 2: Call injections affected the highest number of contracts and airdrop hunting has the highest number of adversarial transactions.

4.2 Manual Analysis

In this part, we perform a manual analysis to filter false positives from our preliminary results and estimate false negatives that are missed in our study.

4.2.1 Methodology and Metrics

Our methodology of manual analysis is as follows. We asked three non-author domain experts to manually review whether unique, non-duplicate contracts are vulnerable and then execute a selected number of unique adversarial transactions of each contract. Domain experts are provided with collected datasets and open-source tools [30,31,33,41] in the validation of contracts. They can also inspect each candidate transaction, e.g., whether transactions have triggered multiple successful `Distr` or `Airdrop` events for the airdrop hunting case. One thing worth noting is that domain experts cannot determine whether 58 closed-source contracts with call injection attacks, not included in Table 2, are vulnerable. Since the total ether loss of those closed-source contracts is less than ten and the total token loss is ignorable, we decide to exclude them from our study.

We adopt three metrics in evaluating our manual analysis, which are evaluation time, agreement rate and Fleiss' kappa [14]. The first is a standard evaluation of how long it takes for human experts to perform all the work, the second is the percentage of analyzed contracts that all three experts consider as vulnerable, and the last a widely-accepted coefficient to measure inter-rater reliability for qualitative data. Three experts take around 30 hours each to evaluate 1,272 contracts and achieve 96.78% agreement rate and 96.47% Fleiss' kappa. Note that many ERC20 token contracts are similar to each other, which greatly reduces human effort.

In 3.22% of cases there were disagreements among human experts, and we asked them to discuss their labeling criteria. In all cases they reached an agreement after the discussion. Here is one example: An ERC 20 contract has an integer overflow vulnerability, but adversarial transactions are targeting another integer underflow vulnerability. One domain expert labels it as true positive and the other two as false positive: After discussion, they agreed on false positive for this example.

4.2.2 Manual Filtering of False Positives

We manually analyze all the transactions and contracts in our preliminary results to find and filter false positives. Table 2 shows the number and rate of false positives and also true positives after filtering. We mainly have false positives for three attack categories, i.e., call injection, integer overflow, and honeypot. Let us explain them separately. First, the false positives of call injection come from the usage of on-chain wallet library, where the library proxies sensitive function calls, like ownership change and ether transfer, specified by input data from wrapper contracts. Second, the false positives of integer overflow are that some toy contracts multiply the number of tokens they provide for fun, and therefore our study mistakenly considers the large token transfer as an integer overflow attack. Lastly, we incorrectly report some betting and lottery contracts with only one winner as honeypots.

4.2.3 Manual Estimation of False Negatives

Because there is no ground truth, we have to create a benchmark and estimate the false negatives of our study. Particularly, we contacted the authors of 11 prior works [23, 27–31, 33, 36, 39–41] on detecting smart contract vulnerabilities and obtained eight replies and six datasets with vulnerable contracts as shown in Table 4. We then sample contracts that are reported by prior works but do not have adversarial transactions reported by our work to estimate false negatives as shown in Table 2. Note that we exclude 38.18% of the candidates with only one creation transaction, which apparently is not adversarial. We then ask our domain experts to go through all the transactions of these contracts and estimate false negatives.

Table 3 shows the estimation of false negative. We only have false negatives for 16 pseudo-bank honeypot contracts

Table 2: A summary of vulnerable contracts and adversarial transactions before and after manual filtering of false positives.

Vulnerability	Preliminary Results		False Positives (FPs)				True Positives (TPs) after Manual Filtering		
	# contract	# confirmed atx	# contract	# confirmed atx	% contract	% atx	# contract	# confirmed atx	# attempted atx
call injection	642	2,996	20	286	3.12%	9.55%	622	2,710	1,494
reentrancy	26	1,948	0	0	0	0	26	1,948	32
integer overflow	56	319	6	36	10.71%	11.29%	50	283	1,367
airdrop hunting	198	100,336	0	0	0	0	198	100,336	57
call-after-destruct	228	1,761	0	0	0	0	228	1,761	0
honeypot	156	266	15	29	9.62%	10.90%	141	237	0
Total	1,272	107,610	41	351	3.22%	0.33%	1,231	107,259	2,633

* atx denotes Adversarial Transactions.

Table 3: Manual estimation of false negatives.

Vulnerability	Evaluation Set		False Negatives (FNs)			
	# contract	# atx	# contract	# atx	% contract	% atx
call injection	8	13	0	0	0	0
reentrancy	50	648	0	0	0	0
integer overflow	50	902	0	0	0	0
airdrop hunting	-	-	-	-	-	-
call-after-destruct	50	811	0	0	0	0
honeypot	192	1,100	16	129	8.33%	11.73%
Total	400	4,546	16	129	4.00%	2.84%

* atx denotes Adversarial Transactions; we leave the FN rates of airdrop hunting as "-" because there are no prior works studying this vulnerability.

Table 4: Availability of related researches' results.

Name	Reply?	Data?	Unique Contracts	Data Until
Oyente [31]	✓	✓	7,527	2016-05-05
ZEUS [29]	✓	✓	1,148	2017-03-15
Maian [33]	✗	✗	-	-
SmartCheck [39]	✓	✗	-	-
Securify [41]	✓	✓	12,276	2017-03-04
ContractFuzzer [28]	✓	✗	-	-
Vandal [23]	✓	✓	101,826	2018-08-30
MadMax [27]	✗	✗	-	-
teEther [30]	✓	✓	1,532	2017-11-30
Sereum [36]	✗	✗	-	-
HoneyBadger [40]	✓	✓	282	2018-10-12
Total	-	-	112,570	-

that pretend to provide bank service for users without setting any bonus—this violates our definition of honeypot in providing bonus. Note that interestingly, we also find some false positives in HoneyBadger's dataset [16]: Specifically, HoneyBadger marked 15 contracts as honeypots but indeed users are capable of gaining profits from them. Our manual verification shows that 13 of them are real lottery and roulette contracts and two are incorrectly-configured honeypots in which users can guess the correct password to win.

5 Results

In this section, we discuss our manually-verified measurement results as summarized in the true positives part of Table 2. In the spirit of open science, we openly release our full results, i.e., all the adversarial transactions, in this URL (https://drive.google.com/open?id=1xLssDxYWyKFCwS5HUrQaSex0uwJRSvDi).

In the rest of the section, we first present real-world adversarial transactions against vulnerable contracts in Section 5.1 and then real-world defenses in Section 5.2.

5.1 Real-world Adversarial Transactions

In this subsection, we present our estimation of ether or token losses of adversarial transactions that we find in the Ethereum blockchain. Here is our methodology of estimating such losses based on different attack categories.

- Reentrancy, integer overflow and airdrop hunting: We get the raw data of ether/token losses by adding up the *absolute* profits and subtracting the cost of the attacker for each transaction.
- Call injection: The call injection attacks we find lead to the ownership change of contracts. Our estimation is to sum up all the ethers or tokens transferred by attackers after the ownership changes.
- Honeypot: We sum up all the ethers transferred by victims to the honeypot across multiple adversarial transactions.
- Call-after-destruct: We sum up all the ethers transferred to the destructed contracts.

We then estimate the monetary losses based on the historical price of ether on Etherscan [13] and tokens on CoinGecko [11]. Note that we are only able to collect the historical price of 13 tokens among all the 259 involved tokens: The value of the rest tokens is considered as zero in our conservative estimation.

Next, we present our loss estimation from two aspects: well-known incidents that are widely reported in the news and other less-known incidents.

5.1.1 Well-known Attack Incidents

In this part, we describe three well-known attack incidents that happen in the history of Ethereum ecosystem and their corresponding losses in Table 5. We categorize all the losses into two parts: direct and actual. Direct loss means that the number of ether loss due to all the adversarial transactions against the vulnerability; actual loss means the amount after deducting the ethers that are saved due to certain tactics—e.g., hard fork and white hat hacking—deployed during the attack. We will describe more details on white hat hacking in Section 6 and only describe the numbers here.

- *TheDAO.* TheDAO, maybe the most famous attack in Ethereum history, is a reentrancy attack. The total amount of confirmed adversarial transactions against TheDAO contract is huge, equaling 11.8 million ethers. However, because the community adopts a hard fork and many white

Table 5: Ether and monetary losses of well-known incidents.

Incident	# contract	# tx	Loss	
			Direct (Ether / $)	Actual (Ether / $)
TheDAO	1	1,841	1,829,473 / $160,146,744	529,041 / $6,213,195
Parity Wallet Hack	622	2,710	204,851 / $40,700,890	154,999 / $31,009,177
SpankChain	1	8	165 / $37,321	165 / $37,321

* Note that although the actual ether loss of Parity Wallet Hack is less than the one of TheDAO, the monetary loss is higher due to the difference in historical ether price.

hat hackers try to save TheDAO, the actual loss is relatively small. Specifically, we have observed that 7.6 million ethers are saved via white hat hacking. The attackers have transferred 3.6 million ethers to the DarkDAO [1], but all the ethers are mandatorily transferred to WithdrawDAO [2] due to the hard fork [24] in July 2016. The rest (i.e., 529,041 ethers), excluding these saved by hard fork and white hat hacking, is considered as the actual loss.

- *Parity Wallet Hack.* Parity Wallet Hack is a call injection attack, in which the vulnerability is in the Parity Wallet library used by many other contracts. We have observed that 622 contracts using Parity Wallet have been attacked, leading to a total direct loss of around 200K ethers. Similar to TheDAO, whitehat hackers have also saved some losses and the actual loss, according to our analysis, is around 155K ethers.
- *SpankChain.* SpankChain is another reentrancy attack targeting the SpankChain contract, a popular ERC20 token with a market capitalization of $6.3 million in August 2019. The loss is only 165 ethers ($37,321), a relatively small number compared to prior incidents. The reason is that SpankChain adopts multiple pluggable modules and the adversary is only able to compromise one of its many payment contracts, leading to a 165 ether loss. We did not see any saving tactics that have been adopted for SpankChain and therefore the actual and direct losses are the same.

5.1.2 Attacks against Other Vulnerable Contracts

In this part, we describe adversarial transactions that target other contracts beyond well-known incidents in Table 6. Airdrop hunting is the largest with $322K monetary loss due to token loss. The loss of honeypot contracts is relatively small, which only has $80K. We estimate the loss of integer overflow as zero, because we could not find any historical price of tokens involved in adversarial transactions targeting integer overflow. We also break down the losses into ether and token as shown in Table 6. Integer overflow and airdrop hunting do not cause any ether loss due to the nature of the attack; on the contrary, both attacks cause a huge amount of token loss. Reentrancy attacks also cause some token loss, relatively smaller than integer overflow and airdrop hunting. Honeypot and call-after-destruct have the least ether loss.

Next, we break down adversarial transactions into those against known and zero-day vulnerabilities separately.

Table 6: A summary of our results in terms of vulnerable contract (vct), confirmed adversarial transactions (atx) and total loss. Note that we exclude three most famous incidents in Table 5 from this table.

Attacks	Known		Zero-day		Total Loss	
	# contract	# atx	# contract	# atx	ether / token	monetary
call injection	-	-	-	-	- / -	-
reentrancy	18	56	6	36	6,080 / 5.01E+23	$142,945
integer overflow	34	167	16	113	- / 7.79E+79	-
airdrop hunting	-	-	197	100,278	- / 3.59E+28	$322,010
call-after-destruct	154	1,547	74	214	472 / -	$100,102
honeypot	90	148	51	-	427 / -	$80,866
Total	285	1,904	344	100,641	6,979 / 7.79E+79	$645,848

* atx: Adversarial Transactions (we mean confirmed atx in this table and skip "confirmed" due to space limits), "-": we do not observe any in our analysis or cannot estimate. We cannot estimate the monetary loss for integer overflow because we cannot find any historical prices of tokens involved in the adversarial transactions.

Vulnerable Contracts Reported by Prior Works We first describe adversarial transactions targeting contracts reported by prior works. As shown in Table 4 and 6, prior works have found 112,570 vulnerable contracts and 298 of these contracts are indeed attacked in real-world, i.e., with 2,061 adversarial transactions in total. This shows a gap between what has been attacked and what has been detected by prior work.

Call-after-destruct has the highest number (i.e., 154) of attacked contracts and a considerable amount (i.e., 90) of honeypots also attract real-world victims. We did not report any call injection because all the observed call injections belong to Parity Wallet Hack; similarly, no prior works have found any airdrop hunting, thus all are categorized as zero-day.

Zero-day Vulnerable Contracts We describe several zero-day vulnerabilities that are not detected or reported by prior works. Our methodology of verifying zero-day vulnerabilities is in four steps as follows. First, we adopt the same six datasets with vulnerable contracts as shown in Table 4 to exclude known vulnerabilities. For reentrancy, we also check and exclude the new patterns found by Sereum paper. Second, we execute existing open-source tools including Mythril, Maian, Securify and teEther to exclude those that can be detected. Third, we check the CVE database with keywords `Smart Contract` and `Ethereum` to exclude these that are available in the database. Lastly, we exclude the vulnerable contracts that have been publicly reported on their websites if available.

Note that as our obligation of responsible disclosure, we have reported all the zero-day vulnerabilities to the contract authors if available online. Specifically, we search for authors' contact information via three ways: (i) source code and comments, (ii) contract main page on Etherscan, and (iii) Google search with the contract address and name. Finally, we have collected the authors' contact information of 42 vulnerable contracts (out of 285 zero-day vulnerable contracts) and communicated to them regarding the found vulnerabilities. At the same time, we have also reported all the reentrancy and

integer overflow zero-day vulnerabilities to CVE. As CVE does not maintain a vulnerability category for airdrop hunting, we have requested to create a new category.

Our results, i.e., the total number of zero-day vulnerable contracts and corresponding transactions, are shown in the "zero-day attacks" column of Table 6. As stated, because no prior works have studied airdrop hunting, almost all the airdrop hunting vulnerabilities except for one reported incident [7] are categorized as zero-day. We also find many zero-days for well-known vulnerabilities and describe them below.

- *Zero-day Reentrancy.* We find six zero-day reentrancy attacks. The main reasons are twofold. First, these zero-day vulnerable contracts adopt function parameters, objects or even another contract to store contract states rather than basic data types like integer. Existing works—no matter static ones like Securify and Mythril or dynamic ones like Sereum—will miss such state updates due to the inaccuracy in the dataflow analysis. Second, these zero-days are cross-function reentrancy, which cannot be detected by Oyente and ZEUS considering only same-function reentrancy.
- *Zero-day Integer Overflow.* We find 16 zero-day integer overflow vulnerabilities because none of prior works has studied integer overflows in token contracts with multi-transfer functionality, e.g., `batchTransfer` and `multiTransfer` functions as an extension to ERC20 standard. Existing works, i.e., Mythril and ZEUS, which claim to check every arithmetic operation, have coverage problem. Particularly, Mythril leverages heuristics to locate all the functions based on known signatures, which do not contain the aforementioned new multi-transfer functions. ZEUS does not model the Ethereum state, thus being unable to reach these vulnerable functions.
- *Zero-day Honeypot.* We find 51 zero-day honeypots with profits, i.e., those that are missed by HoneyBadger, the only honeypot detection work. There are three major reasons. First, we find 42 zero-day honeypots due to incomplete signatures of HoneyBadger. 38 zero-days are *hidden state update* honeypots according to HoneyBadger's classification. In those honeypots, the owner, i.e, the adversary, pays to change the honeypot password and then withdraws the paid money, but HoneyBadger's signature assumes that a honeypot owner needs to call a password change function without paying any ethers. The rest four zero-days are *hidden transfer* honeypots according to HoneyBadger. The misdetection is because those contracts put the logic of preventing victims from transferring money in an invisible long line as opposed to transferring the bait out as modeled by HoneyBadger. Second, we also find a new class of honeypots with two contracts, called *racing time*, which attract users to save ethers with high interest but only leave a short, or even no time window to withdraw. Lastly, we also find seven honeypots with patterns known to HoneyBadger but out of their detection window.
- *Zero-day Call-after-destruct.* We find 74 zero-day call-after-destruct vulnerabilities: The major reason is that many contract destructions are initiated by the owners, which are not modeled by prior works like Maian and teEther. However, other contracts can still call the destructed contracts on the chain despite that the owner destructs it. Destructed contracts can be divided into two categories. First, users are unaware of the contact destruction and continue to participate, thus leading to a loss of money. Second, when a library contract is destructed, many contracts that rely on the library may continue to pay for their services.

5.2 Real-world Defenses against Adversarial Transactions and Evasions against Defenses

In this subsection, we present all the defenses found by our work and their deployments in real-world contracts. Our dataset comes from 5.8 million open-source contracts from Etherscan, which can be reduced to 57K unique contracts. Once we recognize that a defense is deployed by a contract, we evaluate the effectiveness of the defense in terms of prevented and successful adversarial transactions. Here are the results. Our analysis finds six defense classes with attempted adversarial transactions as shown in Table 7. Without loss of generality, we also collect defense libraries from popular secure smart contract library OpenZeppelin [18] and find that all its defenses are already included in our results. Now let us look at the details of each defense in terms of prevented and successful adversarial transactions.

- *onlyOwner*. *onlyOwner* is a Solidity modifier that checks whether a function caller is the contract owner so as to prevent some over-privileged operations, such as `changeOwner` (which literally changes the contract owner) and `mint` (which changes the current supply of token). *onlyOwner* is a widely-adopted defense used by 2,148,200 contracts, because *onlyOwner* is a general defense that prevents any privilege escalation attacks.

  In practice, we did not observe any adversarial transactions that are prevented directly by *onlyOwner*. The likely reason is that adversaries will not launch an attack given the existence of *onlyOwner*. We do observe 2,691 transactions that evade `onlyOwner`: All such transactions are exploiting the Parity Wallet library vulnerability. Particularly, the adversary circumvents the *onlyOwner* defense by changing the contract owner using a call injection vulnerability.
- *isHuman* or *isContract*. This defense checks the code size of a contract to decide whether the caller is a human or a contract, which serves as a bot detection purpose. The intuition is that the code size of a human is zero and the one of a contract is not. This defense is deployed by 21,672 contracts: 36 are airdrop token contracts and the rest include Fomo3D-like [8] ones that reward participants for guessing a correct secret number and other token contracts which handle contract and human invocations separately.

Table 7: Defense techniques against different attacks.

Defense	Checked Values	# of deployed ct	Target Attack	# of prevented atx	# of successful atx
onlyOwner	*msg.sender* state variable *owner*	2,148,200	privilege escalation*	0	2,691
isHuman isContract	*extcodesize*()	21,672	airdrop hunting	14	887
anotherIsHuman anotherIsContract	*tx.origin* *msg.sender*	3,416	airdrop hunting	3	0
canDistr	state variable *distributionFinished*	2,505	airdrop hunting	21	65,240
nonReentrant	state variable *_guardCounter*	952	reentrancy	77	0
SafeMath	function parameters	3,110,124	integer overflow	1,161	55

* Privilege escalation can be the consequence of many existing attacks, such as call injection. *onlyOwner* is a general defense against such escalation.

We have observed that this defense is successful in preventing 14 automatically-launched adversarial transactions. As mentioned in Section 2.1, this defense can be circumvented via embedding code in the contract's constructor function where the code size equals to zero as the contract has not been constructed yet. We have observed 887 adversarial transactions that evade deployed defenses.

- *anotherIsHuman* or *anotherIsContract*. This defense checks whether the origin of transaction equals the sender of the message to ensure that the message sender is not a slave of another master contract. The defense is deployed by 3,416 token contracts. Since the defense is effective in defending against airdrop hunting, we have observed only three transactions that try, but fail to circumvent the defense. We did not observe any adversarial transactions that can evade this defense.
- *canDistr*. *canDistr* is a defense that checks the total number of distributed airdrops to limit the total amount of loss due to airdrop hunting. Such a defense, deployed by 2,505 contracts, is only effective once the hunted airdrops exceed a certain amount. As expected, due to the nature of this defense, it only prevents 21 adversarial transactions when the total hunted airdrops exceed the limit. On the contrary, there are 65,240 adversarial transactions that succeed in obtaining illegitimate airdrops, i.e., evaded this defense.
- *nonReentrant*. *nonReentrant* is a defense that checks the state variable *_guardCounter* to ensure that the function is only invoked once during a transaction. The intuition is that if the function, e.g., a token or ether transfer, is recursively invoked more than once in one transaction, a reentrancy attack is in place. *nonReentrant* is deployed by 952 contracts to prevent reentrancy attacks.

  We have observed 77 adversarial transactions that are prevented by the *nonReentrant* defense. Because the defense is effective, we did not observe any adversarial transactions that evade this defense.
- *SafeMath*. *SafeMath* is a defense library that provides safe arithmetic operations including addition, subtraction, multiplication and division for Solidity contracts. *SafeMath*, the most widely-adopted defense by 3,110,124 contracts, checks whether the operation results have any integer overflows or underflows.

  We have observed that *SafeMath* is successful in defending against 1,161 adversarial transactions that target integer overflow vulnerabilities. Interestingly, we also observed that *SafeMath* is evaded due to incomplete deployment, i.e., mixed use of *SafeMath* functions and normal arithmetic operations. Note that although prior report [10] warned incorrect implementations of *SafeMath*, we did not observe any adversarial transactions.

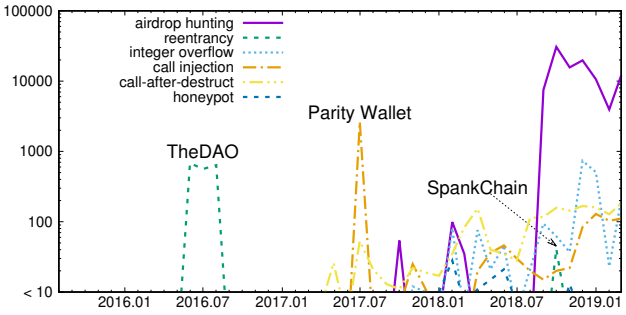


Figure 8: The distribution of adversarial transactions broken down by attack type over time (We marked three well-known incidents in the graph; we excluded all the transactions with a number less than 10 to make the graph clear to view).

6 Observations and Advices

In this section, we describe several observations made from our measurement results and also give some advice to existing security researchers based on our observations.

6.1 Observeations

We present two observations below.

Observation 1 [Attack Strategy Shift]: *The major attack tactics adopted by real-world adversaries evolve from reentrancy in 2016 and call injection in 2017 to honeypot in 2018 and airdrop hunting in 2019.*

Our first observation states that the attack tactics shift over time: For example, Figure 8 shows that after the famous TheDAO and Parity Wallet incidents, the amount of adversar-

Table 8: Peak period of different attack tactics (The average transactions per month of attacked contracts drop significantly after the attack peak).

Vulnerability	Attack Peak	% atx	Average # tx per month	
			Before Peak	After Peak
reentrancy	2016.6 - 2016.8	97.04%	180	0
call injection	2017.7	79.40%	22	0
integer overflow	2018.8 - 2019.1	70.10%	6,007	24
airdrop hunting	2018.8 - 2019.3	82.90%	7,106	167

* atx: adversarial transactions.

ial transactions targeting both reentrancy and call injection drops significantly. We believe that such a shift of attack tactics is due to the following three reasons:

Reason 1.a: *As the smart contract evolves, new attack surfaces are introduced as well.*

The first reason for attack strategy shift is that many new attack tactics are targeting contracts with new features as well. Let us analyze integer overflow and airdrop hunting separately. The burst of integer overflow in 2017 and 2018 is due to the emergence of contracts with multi-transfer functionality, i.e., `transferMulti`, `batchTransfer` and `multiTransfer`, as compared to the old, unprofitable integer overflow vulnerabilities in loop conditions as found by prior works [9, 31].

The burst of airdrop hunting in 2018 and 2019 is because the first airdrop tokens were only spotted in late 2017 and then became much more popular in October 2018. For example, our investigation shows that 851 airdropping contracts have emerged between June 2018 and October 2018. The fast introduction of airdropping contracts leads to corresponding airdrop hunting attacks as well.

Reason 1.b: *The breakout of attacks drains off all the high-value, vulnerable contracts, causing the follow-up attacks unprofitable.*

The second reason for attack strategy shift is that after the breakout of each attack category, e.g., reentrancy and call injection attack, the rest contracts with similar code patterns are of low-value, i.e., being not worth of attacking. Table 8 shows the peak period of each attack category and the average number of transactions of vulnerable contracts before and after the peak. Clearly, the number of users drops significantly after being attacked.

One exceptional category that attackers always like to adopt from its emergence in 2017 until recently is honeypot contracts, the transaction amount of which distributes almost evenly across the time axis. The reason is that there exist no defenses for a honeypot contract, in which the owners are the adversaries waiting for adventurers to fall into the pit.

Reason 1.c: *Most contracts die with very few number of transactions after being attacked.*

We observed a dramatic drop in the number of contracts' transactions the month after being attacked. Over 95% of the attacked contracts suffer an over 75% drop in terms of transactions, and 81.69% contracts have no transactions at all after being attacked. Table 8 shows the average number of normal transactions before and after the attack peak periods broken down by different attack tactics. This clearly shows that most contracts die after being attacked.

We also analyze the contracts that survive the attacks and report the reasons. One typical kind of survivors are airdrop token contracts. The core reason for their survival is that their vulnerabilities are time-limited. Specifically, these token contracts only airdrop newcomers during a "promotion period". Therefore, the amount of airdrop token is actually limited, preventing attackers from draining all the token. The representative of another kind of survivors is SpankChain. Among all the public security incidents, the attacks against SpankChain caused the least loss. Our manual analysis shows that this is due to the modular design of SpankChain contracts. The attacker was only able to compromise one payment channel smart contract of SpankChain, which contains a limited number of ethers compared to its market capitalization (less than 1%). What's more, the modular design enables SpankChain team to apply a quick patch by just replacing the vulnerable payment contract, without affecting other modules.

Observation 2 ["Benign" Adversarial Transactions]: *Some attacks are launched by white hat hackers to save vulnerable contracts.*

Our second observation is that some adversarial transactions, although exploiting a vulnerability, could be "benign" as well. Particularly, many white hat hackers, being aware of the existence of a vulnerability, may exploit it and compete with real adversaries. These white hat hackers will return all the saved ethers and tokens back to the victims afterward. We have observed this trend for both TheDAO and Parity Wallet.

- "Benign" Adversarial Transactions for TheDAO. We observed that 97 adversarial transactions from seven contract addresses, i.e., 5.25% of total adversarial transactions of TheDAO attack, are from "White Hat Group"(WHG) members. All the ethers obtained by these seven addresses—which are 7.9 million ethers—are eventually transferred to a newly-created WithdrawDAO contract [2]. Then, a victim of TheDAO attack can claim those ethers by calling the contract's `withdraw` function, which returns the victim's balance in TheDAO contract back. In August 2019, there still exist 120K ethers in the WithdrawDAO contract.
- "Benign" Adversarial Transactions for Parity Wallet. We observed that 1,959 adversarial transactions, i.e., 72.29% of total adversarial transactions of the Parity Wallet attack, are also from "White Hat Group"(WHG) members, with a total number of 50K ethers. A similar refund contract [3] has also been deployed for victims to get their ethers back.

6.2 Advices

We now present two advices that we are suggesting for smart contract research.

Advice 1 [Improving Existing Program Analysis]: *We sug-*

gest to improve existing program analysis, such as supporting inter-contract dataflow analysis and increasing code coverage.

Our first advice is to improve existing program analysis, because many zero-day vulnerabilities of traditional attack categories are discovered in our study due to the imprecision of prior analysis. For example, several zero-day reentrancies are due to the fact that state variable is stored in a cross-contract location, and zero-day integer overflows are because some code branches, such as new interfaces, are not covered during static analysis. Therefore, we suggest existing security analysis tools using program analysis to include such important factors.

Advice 2 [Keeping Pace with New Strategies]: *We suggest to look for new attack strategies adopted by adversaries and keep pace with corresponding detections and defenses.*

Our second advice is that we should keep looking for new attack strategies adopted by adversaries, such as airdrop hunting studied in this measurement, and then study defenses and detections. Those new attack strategies usually gain very popular within adversaries due to the lack of corresponding defenses and therefore should be of high priority in our research community once emerged.

7 Related Work

In this section, we discuss related work on smart contracts from two aspects: (i) static or dynamic analysis of contracts in detecting vulnerabilities, and (ii) transactional analysis of attacks.

7.1 Smart Contract Vulnerability Detection

Researchers have proposed many works in detecting smart contract vulnerabilities, especially the traditional ones like reentrancy, integer overflow and call injection. On one hand, Oyente [31], Mythril [9], Manticore [5], Vandal [23], Securify [41] and teEther [30] adopt static analysis, e.g., symbolic execution, to detect an execution path between a source, e.g., a contract input, and a vulnerable sink. Bhargavan *et al.* [21] present preliminary work to use existing verification systems to validate smart contracts. ZEUS [29] transfers low-level bytecode to LLVM IR [17] and applies static symbolic model checking to detect whether a violation exists. On the other hand, Sereum [36], a dynamic analysis tool, provides runtime protection against reentrancy attacks and also detects many real-world, new reentrancy patterns that have never been reported before. ContractFuzzer [28] first leverages dynamic testing method [15] to find vulnerabilities in smart contracts. It generates testing inputs from the ABI specifications of contracts and instruments EVM to determine whether vulnerabilities are triggered.

Researchers have also proposed many new vulnerabilities in smart contracts. Maian [33] is the first to propose a class of so-called trace vulnerabilities that have to be triggered by multiple invocations over a contract which belongs to a special type of call-after-destruct. Gasper [25] first points out the extra gas cost caused by under-optimized contract code. MadMax [27] further focuses on EVM gas-focused vulnerabilities, for example, unbounded mass operations leading to an out-of-gas exception. This vulnerability will cause a DoS consequence with no direct ether or token losses. Torres and Steichen [40] observe the rising of honeypot contracts and summarize the techniques adopted by the honeypots.

As a general comparison, our measurement study focuses on these vulnerabilities that are attacked in the real world, but not these contracts that can be attacked. Our study does point out a gap between what has been detected by prior works and what has been attacked—this sets off alarm bells to security researchers on what needs to be studied in the future.

7.2 Transactional Analysis

Researchers have also proposed to analyze smart contract transactions and understand, to some degree, what has been attacked in the past. For example, Sereum replays 78 million transactions involved in the first 4.5 million blocks of Ethereum and confirms reentrancy attacks of two exploited contracts. HoneyBadger crawls all the transactions of 282 true positives to track the ether flowing in and out the honeypots. Perez and Livshits [35] evaluate all the vulnerable contracts reported by prior work and their transactions, and then conclude that only a small number of contracts are actually attacked. We also have a similar observation as Perez and Livshits in terms of attacked contracts.

As a general comparison, our measurement study is the first work to analyze all the transactions during a certain period and study the vulnerability patterns and attack trends. From these attacks, we also evaluate the effectiveness of existing defenses in terms of prevented and successful transactions. Our work sheds a light on future research directions: We believe that researchers should focus on new vulnerability types and old vulnerabilities with new patterns.

7.3 Safer Smart Contracts or Framework

Some researchers are working on the design of a safer smart contract or framework. For example, Breindenbach et al. [22] propose a so-called Hydra Framework, which models and administers bug bounties to incentivize bug disclosure. For another example, Delmolino et al. [26] present their own experience in building safer smart contracts at the University of Maryland. As a comparison, those works are orthogonal to our measurement study, because they are suggesting a new way to develop contracts, while we are studying existing attacks and defenses.

8 Conclusion

In this paper, we perform the first comprehensive measurement study of transactions on the Ethereum blockchain by analyzing real-world attacks and defenses. Our results reveal and quantify the gap between what has been reported by prior

work and what has been adopted by real-world adversaries. Particularly, we have identified 344 previously-unknown vulnerable contracts with 100,641 adversarial transactions.

When analyzing adversarial transactions, we also study how developers adopt defenses against real-world attacks. We have identified six classes of defenses, which prevented 1,276 adversarial transactions in total. We find that these defenses are evaded as well largely due to incomplete deployments. The back-and-forth attacks and defenses show an ever-evolving game that exists in the Ethereum ecosystem.

We are making two suggestions for smart contract related researches. First, we suggest keeping improving prior program analysis to support inter-contract dataflow and increase existing code coverage. Second, we suggest keeping an eye on newly-emerged attack strategies as they can easily draw much attention to adversaries.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper, and the authors of related researches [23,28–31,39–41] for their datasets that help us evaluate our tool. This work was supported in part by the National Natural Science Foundation of China (U1636204, U1736208, U1836210, U1836213, 61972099, 61602121, 61602123), Natural Science Foundation of Shanghai (19ZR1404800). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. This work was supported in part by National Science Foundation (NSF) grants CNS-18-54000 and CNS-18-54001. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

References

[1] TheDarkDAO contract. https://etherscan.io/address/0x304a554a310C7e546dfe434669C62820b7D83490, 2016.

[2] WithdrawDAO contract. https://etherscan.io/address/0xbf4ed7b27f1d666546e30d74d50d173d20bca754, 2016.

[3] ChooseWHGReturnAddress contract. https://etherscan.io/address/0x3abe5285ED57c8b028D62D30c456cA9eb3E74105, 2017.

[4] Ethereum known attacks. https://consensys.github.io/smart-contract-best-practices/known_attacks/, 2017.

[5] Manticore. https://github.com/trailofbits/manticore, 2017.

[6] Parity wallet multi-sig library vulnerability. https://www.parity.io/security-alert-2/, 2017.

[7] Analyzing the first token harvest event in blockchain. https://paper.seebug.org/646/, 2018.

[8] FoMo3Dlong contract. https://etherscan.io/address/0xa62142888aba8370742be823c1782d17a0389da1, 2018.

[9] Mythril. https://github.com/ConsenSys/mythril, 2018.

[10] A redundant SafeMath implementation to make your contract unsafe! https://blog.peckshield.com/2018/08/14/unsafemath/, 2018.

[11] CoinGecko. https://www.coingecko.com, 2019.

[12] Ethereum in bigquery: a public dataset for smart contract analytics. https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics, 2019.

[13] Etherscan. https://etherscan.io, 2019.

[14] Fleiss' kappa. https://en.wikipedia.org/wiki/Fleiss%27_kappa, 2019.

[15] Fuzzing. https://en.wikipedia.org/wiki/Fuzzing, 2019.

[16] HoneyBadger dataset. https://github.com/christoftorres/HoneyBadger/tree/master/results/evaluation, 2019.

[17] LLVM IR. https://llvm.org/docs/LangRef.html#introduction, 2019.

[18] OpenZeppelin contracts is a library for secure smart contract development. https://github.com/OpenZeppelin/openzeppelin-contracts, 2019.

[19] Replay attack. https://en.wikipedia.org/wiki/Replay_attack, 2019.

[20] Solidity programming language: Error handling. https://solidity.readthedocs.io/en/v0.5.11/control-structures.html?highlight=require#error-handling-assert-require-revert-and-exceptions, 2019.

[21] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, 2016.

[22] Lorenz Breindenbach, Phil Daian, Florian Tramèr, and Ari Juels. Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1335–1352, 2018.

[23] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981, 2018.

[24] Vitalik Buterin. DAO fork. https://blog.ethereum.org/2016/07/20/hard-fork-completed/, 2016.

[25] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017.

[26] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine 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, pages 79–94. Springer, 2016.

[27] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. The ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (OOPSLA'18), 2018.

[28] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18), 2018.

[29] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In 25th Annual Network and Distributed System Security Symposium (NDSS'18), 2018.

[30] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In 27th USENIX Security Symposium (USENIX Security'18), 2018.

[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16), 2016.

[32] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. In 9th Annual HITB Security Conference (HITBSecConf), 2018.

[33] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18), 2018.

[34] Santiago Palladino. The parity wallet hack explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/, 2017.

[35] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? arXiv preprint arXiv:1902.06710, 2019.

[36] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In 26th Annual Network and Distributed System Security Symposium (NDSS'19), 2019.

[37] Alex Sherbachev. Hacking the hackers: Honeypots on ethereum network. https://hackernoon.com/hacking-the-hackers-honeypots-on-ethereum-network-5baa35a13577, 2018.

[38] Alex Sherbuck. Dissecting an ethereum honeypot. https://medium.com/coinmonks/dissecting-an-ethereum-honey-pot-7102d7def5e0, 2018.

[39] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB'18), 2018.

[40] Christof Ferreira Torres and Mathis Steichen. The art of the scam: Demystifying honeypots in ethereum smart contracts. In 28th USENIX Security Symposium (USENIX Security'19), 2019.

[41] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18), 2018.

[42] Peter Vessenes. Deconstructing TheDAO attack: A brief code tour. https://vessenes.com/deconstructing-thedao-attack-a-brief-code-tour/, 2016.

[43] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In 27th USENIX Security Symposium (USENIX Security'18), 2018.

A Zero-day Vulnerable Contract Examples

In the appendix, we describe several example zero-day vulnerable contracts in each attack category.

A.1 Zero-day Reentrancy Contract

```
1  contract InstaDice{
2      function payoutPreviousRoll()
3      public
4      returns (bool _success) {
5          ...
6          _finalizePreviousRoll(_user, _stats);
7          stats.totalWon = _stats.totalWon;
8          ...
9      }
10     function _finalizePreviousRoll(User memory
           _user, Stats memory _stats)
11     private {
12         ...
13         require(msg.sender.call.value(_user.
               r_payout)());
14         _stats.totalWon += _user.r_payout;
15         ...
16     }
17 }
```

Figure 9: A zero-day reentrancy contract.

We show the source code of one zero-day reentrancy in Figure 9. The vulnerability is located at Line 13, which has to be triggered through a cross-function call from public interface `payoutPreviousRoll` to private `_finalizePreviousRoll`. Therefore, neither Oyente and ZEUS can detect this cross-function vulnerability. At the same time, the state update is via a membership variable of the function parameter `_stats` at Line 14. Therefore, Securify and Mythril cannot detect the vulnerability.

A.2 Zero-day Integer Overflow Contract

We show the source code of one zero-day integer overflow in Figure 10. The vulnerability is located at Line 6. The contract adopts a vulnerable multiplication operation at Line 6, regardless the deployments of *SafeMath* functions in Line 10 and Line 12. Mythril and ZEUS fail to find the vulnerability because their analysis cannot reach the `batchTransfer` function.

A.3 Zero-day Honeypot Contract

Figure 11 shows a new class of honeypots which attract users to deposit ethers and then refund them. The contract only leaves a one-minute time window for withdrawing, which is hard to satisfy due to the inaccurate timestamp determined by miners. We observed that the contract owner withdrew all the ethers at 7:50 in October, 2011.

B A List of Function Signatures

In this section, we list all the function signatures used by our result analysis of identifying token transfers in Table 9.

```
1  contract PausableToken is StandardToken,
       Pausable {
2      function batchTransfer(address[]
           _receivers, uint256 _value)
3      public whenNotPaused
4      returns (bool) {
5          uint cnt = _receivers.length;
6          uint256 amount = uint256(cnt) * _value
               ;
7          require(cnt > 0 && cnt <= 20);
8          require(_value > 0 && balances[msg.
               sender] >= amount);
9
10         balances[msg.sender] = balances[msg.
               sender].sub(amount);
11         for (uint i = 0; i < cnt; i++) {
12             balances[_receivers[i]] = balances
                   [_receivers[i]].add(_value);
13             Transfer(msg.sender, _receivers[i
                   ], _value);
14         }
15         return true;
16     }
17 }
```

Figure 10: A zero-day integer overflow contract.

```
1  contract Multiple3x is Ownable{
2      uint public refundTime = 1507719600; //
           GMT: 11 October 2017, 11:00
3      uint public ownerTime = (refundTime + 1
           minutes);
4      function refund() payable {
5          require(now >= refundTime && now <
               ownerTime);
6          ...
7      }
8      function refundOwner() {
9          require(now >= ownerTime);
10         if(owner.send(this.balance)){
11             suicide(owner);
12         }
13     }
14 }
```

Figure 11: A zero-day honeypot contract.

Table 9: Sensitive functions related to each result type.

Result Type	Sensitive Function	Signature
token_transfer	transfer(address,uint256)	0xa9059cbb
	transferFrom(address,address,uint256)	0x23b872dd
	transferMulti(address[],uint256[])	0x35bce6e4
	transferProxy(address,address,uint256, uint256,uint8,bytes32,bytes32)	0xeb502d45
	batchTransfer(address[],uint256)	0x83f12fec
	batchTransfers(address[],uint256[])	0x3badca25
	multiTransfer(address[],uint256[]	0x1e89d545
owner_change	setOwner(address)	0x13af4035
	initWallet(address[],uint256,uint256)	0xe46dcfeb
	transferOwnership(address)	0xf2fde38b
	changeOwner(address)	0xa6f9dae1
	addOwner(address)	0x7065cb48