# TXSPECTOR: Uncovering Attacks in Ethereum from Transactions

Mengya Zhang*, Xiaokuan Zhang*, Yinqian Zhang, Zhiqiang Lin
*The Ohio State University*

## Abstract

The invention of Ethereum smart contract has enabled the blockchain users to customize computing logic in transactions. However, similar to traditional computer programs, smart contracts have vulnerabilities, which can be exploited to cause financial loss of contract owners. While there are many software tools for detecting vulnerabilities in the smart contract bytecode, few have focused on transactions. In this paper, we propose TXSPECTOR, a generic, logic-driven framework to investigate Ethereum transactions for attack detection. At a high level, TXSPECTOR replays history transactions and records EVM bytecode-level traces, and then encodes the control and data dependencies into logic relations. Instead of setting a pre-defined set of functionalities, TXSPECTOR allows users to specify customized rules to uncover various types of attacks in the transactions. We have built a prototype of TXSPECTOR and evaluated it for the detection of three Ethereum attacks that exploit: (*i*) the Re-entrancy vulnerability, (*ii*) the UncheckedCall vulnerability, and (*iii*) the Suicidal vulnerability. The results demonstrate that TXSPECTOR can effectively detect attacks in the transactions and, as a byproduct, the corresponding vulnerabilities in the smart contracts. We also show how TXSPECTOR can be used for forensic analysis on transactions, and present Detection Rules for detecting other types of attacks in addition to the three focused Ethereum attacks.

## 1 Introduction

Ethereum is one of the largest public decentralized computing platform built atop blockchain technology. Compared to Bitcoin network [34], Ethereum not only supports simple transactions, but also features Turing-complete computing, in the form of smart contracts. Like many other software programs, smart contracts can be developed using high-level programming languages, such as Solidity [22], and then compiled into bytecode, which are executed in the Ethereum Virtual Machines (EVM) for each node of the peer-to-peer (P2P) network. The capability of executing complex smart contract has become a critical feature of Ethereum compared to the first generation blockchain network.

*These authors contributed equally.

However, greater usability also comes with greater risks. Two features have made smart contracts more vulnerable to software attacks than traditional software programs. (*i*) Smart contracts are immutable once deployed. This feature is required by any immutable distributed ledgers. As a result, vulnerabilities in smart contracts cannot be easily fixed as they cannot be patched. (*ii*) Ethereum is driven by cryptocurrency; many popular smart contracts also involve transfers of cryptocurrency. Therefore, exploitation of smart contracts often leads to huge financial losses. For instance, in the notorious DAO attack, the attacker utilized the re-entrancy vulnerability in *The DAO* contract and stole more than $50 million [27,42]. As another example, a vulnerability in the Parity Multisig Wallet [47] has led to over $30 million losses. Many such attack instances have caused serious concerns regarding the security of smart contracts in Ethereum.

Due to the popularity of Ethereum, efforts have been made to understand and detect these smart contract vulnerabilities such as *re-entrancy*, and *integer overflow* [1], using techniques such as symbolic execution to analyze smart contracts [3,7,31, 43,44] or formal verification to verify its correctness [2,29]. However, using static or symbolic analysis on smart contracts to identify vulnerabilities has its limitations for two reasons. *First*, these tools are difficult to achieve completeness and accuracy simultaneously. For instance, tools using symbolic execution [31,44] suffer from path explosion problems, and existing tools do not detect vulnerabilities involving multiple smart contracts. *Second*, these tools could not be used to inspect and understand real-world Ethereum attacks. Forensic information, such as the pattern and statistics of the attacks, addresses used by attackers, and addresses of victims, can only be learned from transactions. As such, a tool that can perform bytecode-level analysis on the transactions may bring together the best of the two worlds, enabling effective detection and analysis of attacks and vulnerabilities in Ethereum.

In this paper, we present TXSPECTOR, a generic analysis framework for Ethereum transactions to identify real-world attacks against smart contracts in transactions and enable the forensic analysis of the attacks. The key idea of TXSPECTOR is to detect attacks against smart contracts using logic-driven program analysis on Ethereum transactions, and this design is inspired by VANDAL [3], which is a Soufflé-based static

analysis tool for EVM bytecode. The challenges to perform logic-driven analysis on transactions, however, are twofold: *First*, new methods need to be developed to extract data and control dependencies in Ethereum transactions and encode them into logic relations. *Second*, while the number of smart contracts are small, transaction volumes can be huge. Therefore, tracing and analyzing Ethereum transactions requires innovative approaches to optimize the performance.

TXSPECTOR addresses these challenges as follows. *First*, it replays transactions on the blockchain and records bytecode-level traces of the transaction execution. The transaction replay can be achieved all at once or incrementally as new transactions are appended to the blockchain. To avoid repeated efforts, a database of bytecode-level execution traces is built, which can be reused. *Second*, it constructs Execution Flow Graphs (EFGs) to encode the control and data dependencies. *Third*, it extracts logic relations from the EFGs and stores them into databases. *Fourth*, it uses user-specific logic rules (dubbed Detection Rules) to query the databases. TXSPECTOR supports arbitrary Detection Rules defined by users, which enables them to study any aspect of their interests. To the best of our knowledge, TXSPECTOR is the *first* generic framework to perform bytecode-level, logic-driven analysis on Ethereum transactions.

As proof of concept, we apply TXSPECTOR to detect Ethereum attacks that exploit (*i*) the Re-entrancy vulnerability, (*ii*) the Unchecked Call vulnerability, and (*iii*) the Suicidal vulnerability. Our empirical evaluation results on real-world Ethereum transactions show that TXSPECTOR can detect attacks from transactions with a low false positive rate. We also perform a forensic study on the transactions flagged by TXSPECTOR, which reveals several interesting findings of these attacks. In addition to the three focused vulnerability exploits, we also present a number of Detection Rules for readers of interest in Appendix A for other vulnerabilities, such as the Timestamp Dependence vulnerability, the Misuse-of-origin vulnerability, and the FailedSend vulnerability.

**Contributions.** In short, we make the following contributions in this paper:

- **New framework.** We present TXSPECTOR, the *first* generic and logic-driven framework for inspecting the real-world attacks in Ethereum transactions at bytecode level.
- **Comprehensive evaluation.** We evaluate TXSPECTOR's effectiveness in detecting three types of attacks that exploit the corresponding smart contract vulnerabilities.
- **Novel application.** We demonstrate a number of use cases of TXSPECTOR as a forensic analysis tool and perform detailed security analysis on real-world Ethereum transactions.
- **Open source.** To ease the follow-up research for transaction related analysis, we make TXSPECTOR available to the research community under an open-source license at https://github.com/OSUSecLab/TxSpector.

## 2 Background and Related Work

In this section, we first provide the necessary background (§2.1) related to Ethereum including smart contracts and transactions, and then present the corresponding related work (§2.2) to motivate our research.

### 2.1 A Primer on Ethereum Smart Contract

A smart contract is a program of general purpose and executed on a blockchain. It can utilize three memory regions to perform data operations during execution: stack, memory, and storage. A (data) stack is a virtual stack that can be used to store data. Note that EVM also has a call stack, which is different from the data stack. The memory is a byte-addressable region allocated at run-time. Storage is a key-value store that maps 256-bit words to 256-bit words. The stack and memory are both volatile, meaning that the data stored are cleared after each execution. However, the storage is persistent, which can be used to store data across transactions. As a result, the gas price for storage operations are much higher than stack and memory operations.

Currently, EVM supports over 150 OPCODEs [18, 46]. They can be classified into five categories [4] based on the target the instruction operates:

- **Category 1**: OPCODEs that do not operate on any data structures (*e.g.*, `JUMPDEST`).
- **Category 2**: OPCODEs that perform stack operations (*e.g.*, `PUSHx`) or operate on existing values in the stack (*e.g.*, `ADD`).
- **Category 3**: OPCODEs that retrieve information from the blockchain (*e.g.*, `TIMESTAMP`) or the current transaction (*e.g.*, `ORIGIN`).
- **Category 4**: OPCODEs that read/write the memory (*e.g.*, `MSTORE`).
- **Category 5**: OPCODEs that read/write the storage (*e.g.*, `SSTORE`).

Similar to Bitcoin, Ethereum also has a P2P network maintained by Ethereum workers (nodes). To submit a transaction, the user needs to pay a fee called *gas* as an incentive for Ethereum workers to execute the transaction. The *gas* is measured by *Ether*, the cryptocurrency associated with Ethereum. The amount of *gas* needed for each transaction is calculated based on the OPCODEs it includes [46]. If there is not

enough *gas* for executing the transaction, the execution will abort and all the changes will be reverted. However, the *Ether* used during the process will not be refunded. The adoption of *gas* also prevents malicious transactions (*e.g.*, transactions with infinite loops) from jeopardizing the network.

There are two types of accounts in Ethereum: Externally Owned Accounts (EOAs) and Contract (*i.e.*, smart contracts) Accounts [10]. Both types of accounts have the ability to perform Ether transfers. The main difference between them is that smart contracts have the associated bytecode that may be executed, while EOAs do not have any code. In Ethereum, transactions are triggered by EOAs. There are three types of transactions:

- **Type 1**: Transferring *Ether* between EOAs;
- **Type 2**: Deploying a new smart contract on Ethereum;
- **Type 3**: Executing a function of a deployed contract.

The *Ether* transferring transactions do not involve smart contracts; *i.e.*, there is no code execution when processing these transactions. However, to deploy a new smart contract or execute a function of a smart contract, the EVM needs to execute the related bytecode.

Similar to traditional computer programs, smart contracts also contain bugs. Some of them might be exploited by malicious attackers; these bugs are called *vulnerabilities*. There are a number of vulnerabilities identified on smart contracts [1]. To take advantage of these vulnerabilities, attackers need to craft smart contracts and issue transactions targeting the vulnerable ones. Therefore, *attacks* are related to specific transactions, while the root causes of the attacks are the *vulnerabilities* of smart contracts.

## 2.2 Related Work

**Analysis of transactions.** Very few prior studies have performed transaction-based security analysis on Ethereum [24, 37, 38]. SEREUM [38] performs dynamic taint tracking during the execution of transactions to detect a variety of re-entrancy attacks (*e.g.*, cross-function re-entrancy, delegated re-entrancy, and create-based re-entrancy). It only focuses on re-entrancy attacks, which motivates TXSPECTOR to support customized Detection Rules for the detection of various other attacks. ECFCHECKER [24] is another transaction analysis tool that detects if the execution of a smart contract is Effectively Callback Free (ECF), a property that holds for smart contracts that are not vulnerable to re-entrancy attacks. The focus of ECFCHECKER is the re-entrancy vulnerability in smart contracts, while TXSPECTOR is a tool to uncover attacks in transactions. Perez *et al.* [37] recently performed a survey on 21,270 vulnerable smart contracts and their related transactions. While a Datalog-based approach is also adopted, which has inspired TXSPECTOR, the focus of their study was

| Systems | Tx Order Dependence | State Dependence | Mishandled Exception | Re-entrancy | Restricted Transfer | Failed Send | Unsecured Balance | Misuse-of-origin | Integer Overflow | Suicidal | Denial-of-Service |
|---|---|---|---|---|---|---|---|---|---|---|---|
| OYENTE [31] | ▲ | ▲ | ▲ | ▲ | | | | | | | |
| ZEUS [29] | ▲ | ▲ | ▲ | ▲ | | ▲ | | ▲ | ▲ | | |
| SECURIFY [44] | ▲ | | ▲ | ▲ | ▲ | | | | | | |
| VANDAL [3] | | | ▲ | ▲ | | | ▲ | ▲ | | ▲ | |
| GIGAHORSE [23] | | | | ▲ | | | | | | ▲ | ▲ |
| MAIAN [35] | | | | | ▲ | | | | | ▲ | |
| SLITHER [20] | | ▲ | ▲ | ▲ | ▲ | | ▲ | ▲ | | ▲ | |
| MYTHRIL [7] | ▲ | ▲ | ▲ | ▲ | | ▲ | ▲ | ▲ | ▲ | | |
| ETHBMC [21] | | | | | | | ▲ | | | ▲ | |
| SEREUM [38] | | | | ★ | | | | | | | |
| ECFCHECKER [24] | | | | ★ | | | | | | | |
| TXSPECTOR | | ★ | ★ | ★ | | ★ | ★ | ★ | | ★ | ★ |

Table 1: Comparison of TXSPECTOR and related works. ▲: vulnerabilities in smart contracts; ★: attacks in transactions.

the survey of smart contract vulnerabilities; they only analyzed the transactions related to the smart contracts flagged by other tools. TXSPECTOR instead is a tool that performs attack detection and forensic analysis on transactions, which does not rely on smart contracts. Moreover, TXSPECTOR supports customized rules which goes beyond the existing vulnerabilities and attacks.

**Analysis of smart contracts.** Symbolic execution tools, such as OYENTE [31], MAIAN [35], SECURIFY [44], TEETHER [30], MYTHRIL [7] and MANTICORE [43], have been developed to detecting specific vulnerabilities and bugs in smart contracts. While symbolic execution is a powerful approach for discovering bugs, it suffers from the *path explosion* problem and does not scale well. Although not using symbolic execution, SLITHER [20] performs data flow analysis and taint analysis to detect vulnerabilities in solidity programs. SLITHER also suffers from the limitations of other static tools. Closest to ours is VANDAL [3], a static analysis framework extracting logic relations from smart contract bytecode for logic-based analysis. While VANDAL studies the smart contracts, TXSPECTOR analyzes the transactions. To study the dynamic information contained in transactions, TXSPECTOR has to overcome a number of technical challenges (*e.g.*, tracing real values of arguments), which are presented in detail in the following sections. Most recently, ETHBMC [21] was proposed to check the smart contract code using bounded model checking based on symbolic execution. EthBMC can capture inter-contract relations, cryptographic hash functions, and memcopy-style operations, and it can be used to detect suicidal and unsecured balance vulnerabilities.

**Formal verification of smart contracts.** Bhargavan *et al.* [2] presented `EVM*` and `Solidity*`, which can translate smart contract source sode and bytecode into `F*` [40] programs

that can be formally verified. ZEUS [29] is a framework for analyzing safety properties of smart contracts. It translates smart contracts to LLVM IR, adds verification predicates, and feeds them to a verification engine for verification. KEVM [26] is the first fully executable formal semantics of the EVM, which is implemented using the K framework. Park *et al.* [36] extended KEVM and added a few optimizations.

**Summary.** Table 1 compares TXSPECTOR with these related works. While static tools like OYENTE [31] and ZEUS [31] are able to identify one or multiple vulnerabilities in smart contracts, none could be applied to all of them. Moreover, they are not capable of detecting attacks in transactions. Dynamic tools such as SEREUM [38] and ECFCHECKER [24] can detect Ethereum attacks, but they only target re-entrancy attacks. In contrast, TXSPECTOR is capable of detecting various attacks and performing forensic analysis on Ethereum transactions. The dynamic information is crucial in TXSPECTOR, and it is the biggest difference between TXSPECTOR and other static analysis tools. However, TXSPECTOR cannot detect some attacks/vulnerabilities, which will be discussed in §7.4.

## 3 TXSPECTOR Overview

**Objectives.** TXSPECTOR is a software framework for performing logic-driven analysis on Ethereum transactions to uncover attacks and vulnerabilities with three objectives. *First*, it is designed to be a *generic* analysis framework for Ethereum transactions, rather than tailored to detect specific attacks in Ethereum. To this end, it gradually converts transactions into abstractions, without losing important information of the original transactions. *Second*, it is *flexible* and can be extended to analyze transactions in multiple aspects including even non-security related analysis by customizing the Detection Rules. *Third*, TXSPECTOR is also designed to be *performant*. Although it is impossible to perform *generic* attack detection in real-time using the logic-driven framework, efforts have been made to significantly reduce both storage and performance overheads of conducting analysis using TXSPECTOR.

**Scope.** The focus of TXSPECTOR is to detect *attacks* from Ethereum transactions based on the given Detection Rules. Since executing a transaction is basically executing bytecode snippets from multiple smart contracts, TXSPECTOR can also reveal *vulnerabilities* of smart contracts as a byproduct. Therefore, TXSPECTOR is able to identify *attacks* that happened in the blockchain through transactions, as well as the vulnerable smart contracts related to those transactions. However, TXSPECTOR is not designed to detect vulnerabilities in smart contracts, which is aimed by most static analysis tools.

**Overview.** TXSPECTOR consists of four components (Figure 1). Trace Extractor (§4) executes Ethereum transactions and generates bytecode-level traces, which are stored

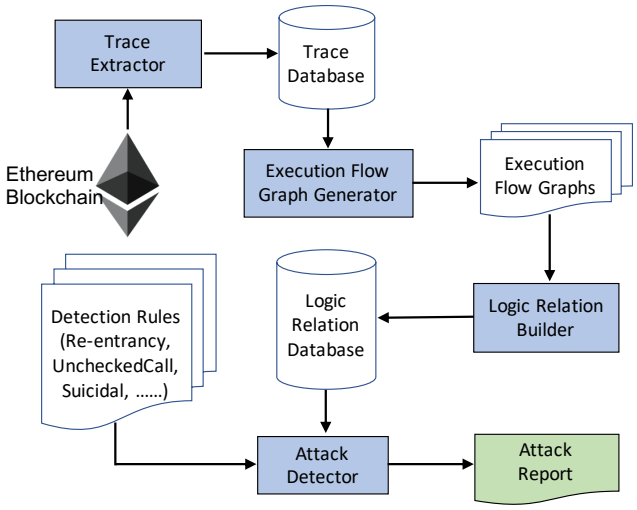


Figure 1: Components and the workflow of TXSPECTOR.

in the Trace Database (DB) for further processing. The bytecode-level traces are then parsed by Execution Flow Graph Generator (§5) for the construction of Execution Flow Graphs (EFGs). Logic Relation Builder (§6) traverses this EFG to extract data and control dependencies and then expresses them into logic relations, which are stored in the Logic Relation DB. Finally, Attack Detector (§7) takes user-specified Detection Rules as inputs to query the Logic Relation DB and outputs the final attack report.

## 4 Trace Extractor

Trace Extractor executes Ethereum transactions inside the Ethereum Virtual Machine (EVM) and records the bytecode-level traces during the execution. A bytecode-level trace is a sequence of 3-tuples. For each OPCODE of the bytecode that is executed by the EVM, Trace Extractor logs its program counter (PC) in the EVM, OPCODE, and its arguments (ARGS) into a 3-tuple, {<PC>; <OPCODE>; <ARGS>}, where the PCs are used to identify OPCODEs by their relative locations in the bytecode. To reduce the data redundancy, Trace Extractor only records the arguments that are not generated from the stack. Because one transaction may indirectly invoke multiple smart contracts, a single bytecode-level execution trace may be generated from the execution of one or more smart contracts. Metadata of the transaction is also recorded, such as the address of the transaction receiver and the timestamp of the transaction.

To replay Ethereum transactions and collect the traces, we modified the `Go-Ethereum` EVM (version 1.8.0) to extract the transaction traces and store them with related metadata in Trace DB. Not all OPCODEs need to be recorded. Type 1 transactions (defined in §2.1) are transactions between EOAs, and there is no bytecode associated with them. Therefore, Trace Extractor only records Type 2 and Type 3 transactions.

**Modifications for `Go-Ethereum` EVM.** To record the transaction traces as shown in Listing 1, we modified the `Go-Ethereum` EVM to log related information of the OPCODEs. More specifically, the modified EVM logs the arguments of the following three types of OPCODEs:

- **Blockchain/transaction related operations (Category 3 defined in §2.1).** This type includes OPCODEs that need to fetch data from the blockchain or the current transaction. For example, `TIMESTAMP` fetches the Unix timestamp of the current block; `CALLER` retrieves the address of the caller.
- **Memory/Storage related operations (Category 4 and 5).** Here, Trace Extractor only records OPCODEs that read data from memory/storage, *i.e.*, `MLOAD` and `SLOAD`. Note that there is no need to record the arguments of `MSTORE` and `SSTORE`, since they only require data from the stack, which can be obtained from other parts of the trace.
- **PUSH operations**. This type includes all PUSH OPCODEs, *i.e.*, $\text{PUSH}_i$, $i = 1, \cdots, 32$.

For the rest OPCODEs, it only records the PC values and the OPCODEs, since there is no need to log the arguments.

**Example traces.** The trace logged by Trace Extractor is similar to the disassembled EVM bytecode. Specifically, a trace is a sequence of 3-tuples, {<PC>; <OPCODE>; <ARGS>}. One transaction trace snippet is shown in Listing 1. The major difference between the traces and the disassembled bytecode is that the recorded traces also contain the real values used in the transaction.

```
 0; PUSH1; 0x60
 2; PUSH1; 0x40
 4; MSTORE
 5; CALLDATASIZE; 0x144
 6; ISZERO
 7; PUSH2; 0x20e
10; JUMPI
```

Listing 1: Trace Snippet

```
 0: V0 = 0x60
 2: V1 = 0x40
 4: M[0x40] = 0x60
 5: V2 = 0x144
 6: V3 = ISZERO 0x144
 7: V4 = 0x20e
10: JUMPI 0x20e 0x0
```

Listing 2: IR Snippet

**Trace DB.** The trace DB stores the recorded bytecode-level traces and related metadata while executing the transactions. Specifically, each trace of a transaction can involve more than one smart contract since multiple smart contracts may be invoked, and the metadata includes the information of (*i*) the transaction sender (*i.e.*, the sending party of a transaction), (*ii*) the transaction receiver (*i.e.*, the receiving party), and (*iii*) the timestamp (*i.e.*, the date and time at which a transaction is included in a block) of the transaction.

## 5 Execution Flow Graph Generator

To express the control-flow more explicitly, Execution Flow Graph Generator builds Execution Flow Graphs (EFGs) that encode the control and data-flow information of the traces into

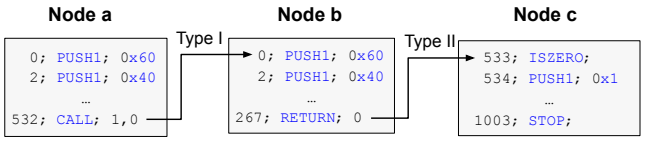


Figure 2: An example of Execution Flow Graph.

graphs. Since the bytecode-level traces are generated from transactions, there is no unresolved branch in the EFG. Therefore, the execution flow is sequential in each smart contract. A node in an EFG represents the execution of one smart contract, which contains the bytecode-level execution trace generated by this contract. An edge in an EFG represents a control-flow transfer from one smart contract to another.

The Execution Flow Graph Generator parses the bytecode-level traces to construct the Execution Flow Graphs (EFG). Since the trace is dynamically generated, there is no *unresolved* branch and each `JUMP` only has one destination. The nodes and edges in the EFGs are created in the following ways:

- **Node.** Execution Flow Graph Generator generates a new node when the execution flow is altered from one Smart contract to another. Specifically, when a `CALL`-related OPCODE is encountered, *i.e.*, `CALL` / `DELEGATECALL` / `CALLCODE` / `STATICCALL` or a `STOP`-related OPCODE is encountered, *i.e.*, `STOP` / `REVERT` / `RETURN`, a new node is generated.
- **Edge.** When the execution flow transfers from one smart contract to another, Execution Flow Graph Generator will generate the edge that represents the control flow between two nodes. There are two types of edges: Type I edge is an edge from a caller contract to a callee contract. Type II edge is an edge from a callee contract to a caller contract.

An EFG example that involves three smart contracts is shown in Figure 2: smart contract A (Node *a*) first calls smart contract B (Node *b*), generating a Type I edge, which transfers the execution flow to smart contract B. When smart contract B finishes execution, it returns to smart contract A (Node *c*), generating a Type II edge. The EFG ends in Node *c*.

To analyze execution traces involving multiple smart contracts, each 3-tuple in the original trace is augmented to a 6-tuple: {<PC>; <OPCODE>; <ARGS>; <`idx`>; <`depth`>; <`callnum`>}. We define `idx`, `depth` and `callnum` as follows:

- **Idx.** Because there are identical PC values in different contracts, it is not possible to tell which OPCODE is executed first solely from their PC values. Therefore, the `idx` parameter is introduced for each opcode to represent the index of the current OPCODE in the EFG.
- **Depth.** When dealing with a trace with a lot of external calls, it is important to know which call-level each

OPCODE is in. To this end, we introduce `depth`, which describes the call depth of each OPCODE in an EFG. Whenever there is a call-related OPCODE encountered, the `depth` increases by 1; when it returns, the `depth` decreases by 1.

- **Callnum.** The `callnum` represents the number of calls happened before each OPCODE in the EFG. It is a non-decreasing value: it increments by 1 when encountering a call-related OPCODE.

## 6 Logic Relation Builder

The Logic Relation Builder first parses the EFGs to construct intermediate representation (IR) suitable for our analysis, then extracts the logic relations that express the semantics of the transactions by defining logic rules. After that, the logic relations are stored in the database. Particularly, logic rules are defined to express control-flow and data-flow information, in order to obtain the control and data dependencies in transactions. For instance, some rules dictate the execution order of opcodes, which is related to the control-flow; some rules track how arguments of OPCODEs are defined and used, which is related to the data-flow. To achieve this, the Logic Relation Builder generates logic relations for each OPCODE, such as the registers representing their operands, and their *PC* values. Meanwhile, it associates the real values in the transaction with the registers, so that the dynamic information is captured. As such, the control and data dependencies are encoded into logic relations, which are then organized and stored in the database.

**Converting Trace-based EFG to IR.** TXSPECTOR adopts the IR specification in VANDAL [3]. This IR is a register-based language, which is another form of expressing data and control dependencies. IR replaces the stack operations with registers. For example, the corresponding IR of the example trace snippet in Listing 1 is shown in Listing 2. We thus have extended VANDAL in the following two aspects: *First*, we need to deal with real values rather than symbolic ones. This is achieved in Logic Relation Builder by simulating the EVM stack operations using the registers with real values, so that the values of registers are updated accordingly, and all of the intermediate values are properly recorded. This is a crucial step to capture all the dynamic information during transaction execution, which cannot be achieved by static analysis tools. For example, when processing `TIMESTAMP`, the real timestamp value recorded in the bytecode-level trace is pushed into stack and assigned to its related register. *Second*, we need to deal with inter-contract calls. For Type-I edges in the EFG, the current stack is sealed and an empty stack is created. For Type-II edges in the EFG, the current stack is deleted and the last sealed stack is resumed.

```
1  .type Variable
2  .type Opcode
3  .type Value
4  .decl def(var:Variable, pc:number, idx:number,
     ↪  depth:number, callnum:number)
5  .decl use(var:Variable, pc:number, i:number, idx:number,
     ↪  depth:number, callnum:number)
6  .decl op(pc:number, op:Opcode, idx:number)
7  .decl value(var:Variable, val:Value)
8  .decl op_OPCODE(pc:number, registers:Variable, idx:number,
     ↪  depth:number, callnum:number)
9  .input def, use, op, value, op_OPCODE
```

Figure 3: The logic rules used by Logic Relation Builder.

| PC | Register | Idx | Depth | Callnum |
|---|---|---|---|---|
| 0 | V1 | 1 | 1 | 0 |
| 2 | V2 | 2 | 1 | 0 |
| 0 | V89 | 245 | 2 | 1 |
| 2 | V90 | 246 | 2 | 1 |
| 534 | V285 | 1,072 | 1 | 1 |

Table 2: An example of `PUSH1` logic relations.

**Generating logic relations from IR.** Inspired by VANDAL, TXSPECTOR adopts and extends its logic rules to deal with real values and traces with multiple smart contracts. The rules used in TXSPECTOR are shown in Figure 3. For example, the relation *op* associates an OPCODE with a `pc` and its `idx`. The real values used in a transaction are extracted by the *value* relation, which records the registers and the related values. Every OPCODE and its related registers are also extracted into logic relations. For example, the logic relation of `SSTORE` documents all `SSTORE`s in the EFG and the tuples $(\{pc, registers, idx, depth, callnum\})$ related to them.

One example of the logic relations is listed in Table 2, which represents the `PUSH1` OPCODE from the EFG shown in Figure 2. In particular, Row 1 and 2 in the table come from Node *a*; row 3 and 4 come from Node *b* and their `depth` has been changed from 1 to 2 and `callnum` from 0 to 1. Row 5 comes from Node *c*. Its `depth` has changed from 2 to 1 since the call returns, but its `callnum` remains the same as the number of calls has not changed.

## 7 Attack Detector

Attack Detector is the key component of TXSPECTOR that takes user-specified query rules (dubbed Detection Rules) as inputs and queries the Logic Relation DB generated by Logic Relation Builder to reason about a specific security property of the transactions. Once the Logic Relation DB is generated, it can be used for different types of analysis; there is no need to reconstruct a new DB for every Detection Rule. The outputs are not simple *yes* or *no* answers for a specific query; instead, detailed information regarding the attacks, if detected, is also provided to allow further analysis.

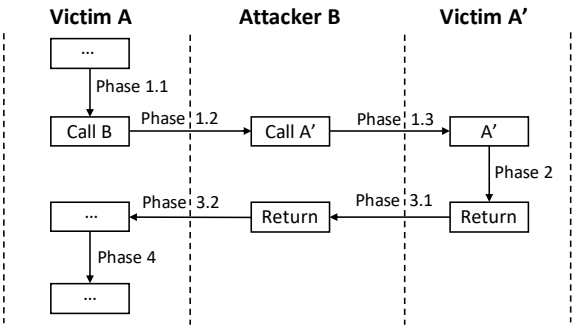


Figure 4: An example of *inconsistent state*.

We choose to build Attack Detector using `Soufflé` [28], which is a state-of-the-art Datalog query tool with high performance. Therefore, in TXSPECTOR, Detection Rules are written in `Soufflé`, which is a variant of the `Datalog` language. In this section, we show how to construct Detection Rules to detect attacks in transactions using three examples: *Re-entrancy*, *UncheckedCall*, and *Suicidal*. We also present three other Detection Rules for detecting *Timestamp Dependence*, *Misuse-of-origin* and *FailedSend* attacks in the Appendix A for readers of interest.

## 7.1 Rules for Re-entrancy Attacks

**Description.** Being one of the most severe attacks, re-entrancy attack targets the Ethereum smart contracts since the re-entered smart contract may transfer Ether multiple times. When contract A calls contract B, contract B may re-enter contract A again in the same transaction. If contract B is malicious, it may take advantage of contract A's intermediate state (*e.g.*, obsolete account balance) to steal *Ether* from A. This type of attacks is called re-entrancy attacks, since they are caused by re-entering the caller contract (contract A) in the same transaction. The most infamous re-entrancy attack is the DAO attack [27,42], in which the attacker stole a large amount of *Ether* (worth over $50 million) from the DAO contract.

Our goal is to design Detection Rules to detect the advanced re-entrancy attacks mentioned in SEREUM [38]. If there is a state change (*i.e.*, updates of a storage variable) after the Victim Contract is re-entered and returned, and this storage variable affects a control-flow decision when re-entering the Victim Contract, it will result in an *inconsistent state*.

**Requirements.** Suppose that contract A is the *Victim* and contract B is the *Attacker*. We define four phases:

- **Phase 1:** A executes its code (*Phase 1.1*) and calls B (*Phase 1.2*); B calls A again to re-enter A (*Phase 1.3*). The re-entered A is denoted as A';
- **Phase 2:** A' executes its code, before returning to B;
- **Phase 3:** A' returns to B (*Phase 3.1*), and B returns to A (*Phase 3.2*);
- **Phase 4:** A continues its execution.

The transaction should at least have all these four phases in order to perform a re-entrancy attack, and having at least one *inconsistent state* is a necessary condition of a re-entrancy attack. An example with the four phases is shown in Figure 4. Given the four phases, there are two requirements for the *inconsistent state*: (*i*) **`SLOAD-JUMPI` dependency:** In *Phase 2*, A' loads (using `SLOAD`) a storage variable ($V$) for a control-flow decision (*i.e.*, condition for a `JUMPI` instruction); (*ii*) **`SLOAD-SSTORE` dependency:** In *Phase 4*, A updates (using `SSTORE`) the storage variable $V$. If a transaction satisfies both requirements, it is clear that in *Phase 2*, A' loads $V$ from an *inconsistent state* for the control-flow decision. As a result, B is able to manipulate the control flow by re-entering A, thereby launching a re-entrancy attack.

**Detection Rules.** We define our Detection Rules based on the requirements for the *inconsistent state*. More specifically, we define the following Detection Rules (shown in Figure 5):

The Detection Rules first check the `SLOAD-JUMPI` Dependency. If a value loaded by the `SLOAD` (*sloadVal*) is used in the condition of a `JUMPI`, the `SLOAD` address (*sloadAddr*) is obtained. The `SLOAD` and `JUMPI` should have the same `depth` and `callnum` (defined in §5), and the condition of `JUMPI` (*jumpiCond*) should depend on the value of `SLOAD` (*sloadVal*), which is enforced by the **depends** Detection Rule. The **depends(A, B)** Detection Rule checks whether there is a data flow from A to B.

Next, the Detection Rules check the `SLOAD-SSTORE` Dependency. First, they check whether there is an `SSTORE` working on an address (*sloadAddr*). If so, they check (*i*) whether this address is already used by an `SLOAD` that satisfies the first condition via the **checkSameAddr** Detection Rule, (*ii*) whether the `SSTORE` is executed after the `SLOAD`, and after an external call returns (by checking the `idx` and `depth` via the **filterByDepth** Detection Rule and the **filterByIdx** Detection Rule, respectively). The **checkSameAddr** Detection Rule checks whether `SSTORE` and `SLOAD` have the same address. The **filterByDepth** Detection Rule keeps `SLOAD` and `SSTORE` pairs in which *sloadDp* is larger than *sstoreDp*. The **filterByIdx** Detection Rule keeps `SLOAD` and `SSTORE` pairs where *sloadIdx* is less than *sstoreIdx*. In addition, they check whether `SLOAD`, `SSTORE`, and `JUMPI` are in the same contract (via the **checkSamecontract** Detection Rule). If so, then an *inconsistent state* is detected, which indicates a re-entrancy attack.

## 7.2 Rules for UncheckedCall Attacks

**Description.** An *UncheckedCall* attack can be exploited to steal Ether [25] due to the lack of checks on the return value of an external call. Specifically, in Ethereum smart contracts, the `CALL` OPCODE is used frequently for inter-contract communications and cryptocurrency transfers (*i.e.*, `send` function).

```
Reentrancy(args):-
  % SLOAD-JUMPI Dependency
  op_SLOAD(_, sloadAddr, sloadVal, sloadIdx, sloadDp, cn),
  op_JUMPI(_, _, jumpiCond, _, sloadDp, cn),
  depends(jumpiCond, sloadVal),

  % SLOAD-SSTORE Dependency
  op_SSTORE(_, sstoreAddr, _, sstoreIdx, sstoreDp, _),
  filterByDepth(sloadDp, sstoreDp),
  filterByIdx(sloadIdx, sstoreIdx),
  checkSameAddr(sloadAddr, sstoreAddr),
  checkSameContract(sloadAddr, jumpiCond, sstoreAddr).
```

Figure 5: The Detection Rules for detecting Re-entrancy.

During an external call, exceptions might happen, which will cause the callee contract to revert its execution and return. Ideally, the caller should check the return value of the call. If it is zero (*e.g.*, caused by exception during the call), it should take actions (*e.g.*, revert its execution) to handle the exception properly. However, many developers do not perform such checks. As a result, these contracts have the *UncheckedCall* vulnerability and cause the money stolen.

**Requirements.** We adapt the detection criteria of a bytecode analysis tool, SECURIFY [6, 44], to define the requirements of *UncheckedCall* attacks. The transaction that contains the *UncheckedCall* attack should meet the following requirements: (*i*) **External call:** There is at least one external call (`CALL`-related OPCODE) in the transaction. (*ii*) **Unchecked call return value:** There is at least one external call whose return value is not used by any `JUMPI`.

Having at least one unchecked return values is a necessary and sufficient condition of an *UncheckedCall* attack. To be more specific, at bytecode-level, checking the return value is done with a `JUMPI` depending on the return value of the `CALL`. Therefore, if in a transaction, there is the call return value not used by any `JUMPI`, it means that this value is not checked and this transaction is under *UncheckedCall* attack.

**Detection Rules.** To detect the *UncheckedCall* attack, we define our Detection Rules in Figure 6. The *UncheckedCall* Detection Rules first extract all the call return values in a transaction (line 7), and check whether there is a `JUMPI` depending on each of the call return values, using the **jumpiDep** Detection Rule. If there is a call return value not being used by any `JUMPI`, the transaction is flagged as *UncheckedCall*. Note that a transaction may include OPCODEs from multiple contracts. The `depth` of both `CALL` and `JUMPI` in our Detection Rules are set to 1, so that only the *UncheckedCall* attack targeting the Receiver contract is detected.

## 7.3 Rules for *Suicidal* Attacks

**Description.** A "*Suicidal*" attack can cause the smart contract killed by anyone, rather than the contract owner, due to the lack of proper permission check. Specifically, Ethereum provides smart contracts with the ability to remove themselves from the blockchain via the `SELFDESTRUCT` OPCODE. While

```
jumpiDep(jumpiIdx, jumpiDepth, depIdx, depVal) :-
  op_JUMPI(_, _, jumpiCond, jumpiIdx, jumpiDepth, _),
  jumpiIdx > depIdx,
  depends(jumpiCond, depVal).

UncheckedCall(args) :-
  op_CALL(_, _, _, callRet, callIdx, 1, _),
  !jumpiDep(jumpiIdx, 1, callIdx, callRet).
```

Figure 6: The Detection Rules for detecting *UncheckedCall*.

```
Suicidal(args) :-
  op_SELFDESTRUCT(_, _, sdIdx, 1, _),
  op_CALLER(_, callerAddr, callerIdx, 1, _),
  !jumpiDep(jumpiIdx, 1, callerIdx, callerAddr).
```

Figure 7: The Detection Rules for detecting *Suicidal*.

the design of `SELFDESTRUCT` is for contract owners to manage the life cycles of their smart contracts, some smart contracts fail to add proper permission checks before calling `SELFDESTRUCT`. Since TXSPECTOR examines transactions instead of smart contracts, it detects the attacks where unauthorized users trigger the `SELFDESTRUCT` of smart contracts. Each contract can at most be detected *once*, as it has been destroyed afterwards.

**Requirements.** The *Suicidal* attack can be detected by checking whether there is a permission check (*i.e.*, `JUMPI`) before executing `SELFDESTRUCT`. There are two requirements: (*i*) **`SELFDESTRUCT`:** There is at least one `SELFDESTRUCT` in the transaction. (*ii*) **No permission check on `CALLER`:** There is no `JUMPI` that depends on `CALLER`.

If there is no `CALLER-JUMPI` dependency in a transaction, it means that it does not check the `msg.sender` (`CALLER`) before executing `SELFDESTRUCT`, which further indicates the contract can be killed by anyone, *i.e.*, a *Suicidal* attack.

**Detection Rules.** The Detection Rules to detect the *Suicidal* attack are shown in Figure 7. The Detection Rules first make sure that there is at least one `SELFDESTRUCT` in the transaction, then examine whether the *msg.sender* is checked before `SELFDESTRUCT` via **jumpiDep** Detection Rule (line 3-4).

## 7.4 Rules for Other Attacks

Besides attacks exploiting the three vulnerabilities mentioned above, we also demonstrated the use of TXSPECTOR to detect other types of attacks, such as the *Timestamp Dependence* (§A.1), the *Misuse-of-origin* (§A.2) and the *FailedSend* (§A.3). However, not all known attacks/vulnerabilities can be detected by TXSPECTOR. For instance, the *transaction order dependence* involves multiple transactions, but TXSPECTOR performs analysis on a single transaction; the *restricted transfer* is not observable in transactions; detecting the *integer overflow/underflow* requires source code level information (*e.g.*, types), which is missing in the bytecode. Note that we have summarized these attacks/vulnerabilities in Table 1.

## 8 Evaluation

In this section, we first explain the setup of our evaluation, and evaluate TXSPECTOR when applying the 3 Detection Rules mentioned in §7 to detect attacks in transactions.

### 8.1 Experiment Setup

**Trace collection.** The traces were collected on an L8s v2 instance on the Microsoft Azure Cloud [32], with 8 VCPUs, 64GB RAM and 2TB SSD, running Ubuntu 18.04. Trace Extractor ran a full Ethereum node to collect bytecode-level traces, from the 0-*th* block to the 7,200,000-*th* block. Note that Ethereum has around 10180000 number of blocks (as in June 1st 2020) and it keeps growing exponentially. We cannot collect all of them for our experiment because it takes a huge amount of storage and also processing time. We therefore stop collecting at 7,200,000-*th* block, which has resulted in the size of 1,577 GB, containing 397,269,533 transactions in total. We store them in a Trace DB (implemented atop MongoDB [33]) by Trace Extractor.

**Dataset.** Having collected the traces of the transactions of blocks, we then derive the transactions from them, which are the input to TXSPECTOR. Given the huge volume of blocks we have, we cannot take all of them to derive the transactions because a block may contain multiple transactions. We therefore decide to only focus on the transactions starting from the 7,000,000-*th* block as our dataset, which contains 16,485,279 transactions, covering the transactions between January 2019 to February 2019. With such 16 million transactions, we believe it is representative to cover various situations for our experiment.

**Logic relation generation.** Our dataset related to logic relation generation contains 9,661,593 transactions, which is acquired through two steps. *First*, the dataset originally contained 16,485,279 transactions. However, not all transactions have traces; that is, they do not invoke the execution of smart contracts. We have to filter them out, because our logic relation generation process only takes transactions with trace as inputs. After filtering, there were 9,662,675 transactions left. *Second*, the raw traces were processed to generate the logic relations. However, not all the transactions can be processed due to the timeout. We therefore set the timeout threshold to be 60 seconds and processed each of these traces through the Execution Flow Graph Generator and the Logic Relation Builder to generate the logic relations. Unfortunately, 1,082 transactions (0.01%) did not finish logic relations generation on time. As such, eventually our final dataset contains 9,661,593 transactions. The Logic Relation DB takes 2,949 GB space.

The majority of the logic relations is generated in a very short time window. Specifically, there are only 94,277 (1.0%) transactions that have a processing time larger than 4*s*. We plotted the processing time distribution of the transactions

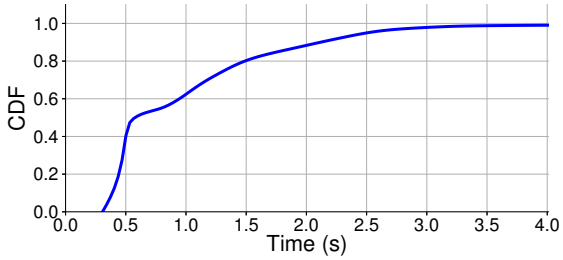


Figure 8: Time distribution on generating logic relations.

that finished processing within 4s (99.0% of the 9,662,675 transactions) in Figure 8. About 60% transactions finished generating logic relations within 1*s*. If we set the timeout threshold to 2*s*, logic relations of about 90% transactions can be generated. It took 1.03*s* on average to generate the logic relations for one transaction. Note that for each transaction, we only need to generate the logic relations once, no matter how many Detection Rules to be applied in the Attack Detector, since different Detection Rules use the same logic relations.

After generating the logic relations, we applied the Detection Rules to detect attacks and vulnerabilities in transactions. The timeout threshold was set to 1*s* in the Attack Detector. We studied the reasons of the Logic Relation timeouts and the Detection Rule timeouts in §8.6.

**Evaluation steps and criteria.** We made several steps during the evaluation and also compared our results other static analysis tools. We first applied TXSPECTOR to flag the transactions based on the Detection Rules. For the flagged transactions, if the source code of the receiver smart contract is available, we then performed manual inspection to check if they are vulnerable to the specific attack. Next, if a smart contract is vulnerable, the flagged transactions related to these contracts are considered true positives. Otherwise, the related transactions are considered false positives. Due to the large volume of transactions, we have no means to analyze negative results. The same issue was also faced by other related works (*e.g.*, SEREUM). Finally, we also compare TXSPECTOR with three Datalog-based static analysis tools: SECURIFY, VANDAL and GIGAHORSE, if the specific vulnerabilities can be detected by these tools.

### 8.2 Results of Re-entrancy Attacks

First, we present the detection results of re-entrancy attacks. We applied the re-entrancy Detection Rules to the 9,661,593 transactions in our dataset. 336,909 transactions (3.5%) did not finish due to timeout. For the 9,321,684 transactions that had a verdict, TXSPECTOR flagged 3,357 transactions (0.04%) as re-entrancy attacks. These 3,357 transactions were related to 30 smart contracts, and 22 of them were open-source. We decompiled the 8 closed-source contracts using the online Solidity decompiler [19]. After our manual inspection, we confirmed that 10 of the 22 open-source contracts and 7 of the 8 closed-source contracts contained re-entrancy

vulnerabilities. There are two main reasons why TXSPECTOR mis-flagged 13 smart contracts: (*i*) It fails to detect the lock that prohibits unauthorized reentering the re-entrant function; (*ii*) The mis-flagged contracts can be re-entered, but it is not possible to steal Ether or token from them. One example of false positives is transaction `0xd32496` [17]. A code snippet of the related functions is shown in Listing 3, which uses a lock to prevent unauthorized re-entrancy attempts. The lock variable (*reentrancyLock*) is checked before the call and updated after the call, which allows the contract to be re-entered only once. As such, a practical attack is prevented, but the execution of the transaction meets our requirements of re-entrancy attacks (§7.1). As a result, it is a false positive.

```
function nonReentrant() {
    require(!reentrancyLock);
    reentrancyLock = true;
    call(...);
    reentrancyLock = false;
}
```

Listing 3: A false positive example of re-entrancy.

The detection results are on par with SEREUM [38], which flagged incorrectly 46,743 out of 49,080 transactions (among the total 77,987,922 transactions), rendering a false positive rate of 0.06%. Following the same criteria, TXSPECTOR mis-flagged 3,072 transactions (related to the 13 smart contracts), yielding a false positive rate of 0.03%. However, we believe these values are only approximation of the detection accuracy, as it is impossible to count true negatives.

In order to compare the detection results of TXSPECTOR with other tools, we either reached out to the corresponding authors for help and clarification, or ran the open-sourced tools with the same dataset. The results are summarized in Table 3. We present the comparisons in detail below.

**Comparison with SEREUM.** There is no open-sourced release of SEREUM; but we reached out to the authors of SEREUM and obtained their evaluation result for comparison purposes. For the same dataset, SEREUM flagged 10,278 transactions as re-entrancy attacks, 2,732 of which were also marked by TXSPECTOR. For the remaining 7,546 transactions, we found that 7,271 of them did not have a result in our dataset due to timeout[1]. There are 625 transactions that are flagged by TXSPECTOR but not SEREUM. While our manual inspection suggests they lead to *inconsistent state*, we do not understand why they are not identified by SEREUM.

**Comparison with SECURIFY.** We performed a comparison with static analysis tool SECURIFY. Note that SECURIFY aims to detect re-entrancy vulnerabilities on smart contracts, while our focus is the transaction. We first extracted all receiver smart contracts of the transactions in our dataset and then applied SECURIFY on these smart contracts. The total number of receiver smart contracts in our dataset is 105,535. For the 3,327 transactions flagged by TXSPECTOR, there were only 30 receiver smart contracts. We ran the open-sourced version of SECURIFY [14] on the 105,535 receiver smart contracts. The timeout threshold is set to 60*s* for analyzing each contract. 1,315 of them did not finish due to timeout; 6,226 of them did not have result due to run-time errors. For the remaining 97,994 smart contracts, SECURIFY flagged 1,196 of them as re-entrancy, and none of them were flagged by TXSPECTOR.

After reading the source code of SECURIFY, we found that it defined two kinds of re-entrancy, "Gas-dependent Reentrancy" [12] and "Reentrancy with Constant Gas" [13]. But in our definition, we check the *inconsistent state*, which requires the state update after call. Therefore, we conclude TXSPECTOR leads to different detection results from SECURIFY as they have a different criterion of detecting re-entrancy. Detection Rules can also be defined to detect these types of re-entrancy attacks using TXSPECTOR (details are in §A.4). It is worth noting that SEREUM [38] also mentioned that "Securify defines a very conservative violation pattern for re-entrancy detection that forbids any state update after an external call" and, as a result, leads to "a very high false positive rate".

**Comparison with VANDAL.** We used the open-sourced version of VANDAL [45] for comparison. The timeout threshold was set to 60*s* for analyzing each contract. When analyzing the 105,535 receiver smart contracts, 1,206 of them (1.1%) did not finish within 60*s*; 225 (0.2%) did not have result due to some runtime errors. For the remaining 104,104 smart contracts, VANDAL flagged 85,721 (82.3%) as reentrant, which is clearly not reasonable. We randomly selected some of the detected smart contracts and found they were all false positives. Because the number of flagged contracts are huge, we cannot perform manual inspection on all of them.

By checking the rules provided by VANDAL, we found that the rules are much more relaxed than ours. According to their paper, "*A call is flagged as reentrant if it forwards sufficient gas and is not protected by a mutex*". As a result, any call with sufficient gas and no lock will be marked as reentrant by VANDAL, which is a much relaxed criterion. Among the 30 smart contracts marked by TXSPECTOR, 27 were also flagged by VANDAL. For the remaining 3, VANDAL did not finish analyzing them due to timeout. Therefore, TXSPECTOR outperforms VANDAL in that it leads to low FP rate.

**Comparison with GIGAHORSE.** There is no open-source release of GIGAHORSE, but there is a website [8] for users to query the results of GIGAHORSE. We extracted all results of the "Reentrancy" from their website. Among the 105,535 receiver smart contracts in our dataset, 3,310 (3.1%) of them are flagged as reentrant by GIGAHORSE. 18 out of 30 smart contracts detected by TXSPECTOR are also flagged by GIGAHORSE; the remaining 12 are not considered vulnerable by GIGAHORSE. According to the explanation on the FAQ

[1] We believe because there are deep recursions in re-entrancy transactions, their traces are extremely long and complex, and have a higher probability of causing timeouts in TXSPECTOR. We analyze the reasons of timeouts in §8.6.

| Vulnerability | System | # Total | # Timeout or Error | # Remaining | # Flagged |
|---|---|---|---|---|---|
| Reentrancy | TXSPECTOR | 9,661,593 | 336,909 | 9,321,684 | 3,357 |
| | SEREUM | 9,661,593 | N/A | N/A | 10,278 |
| | SECURIFY | 105,535 | 7,541 | 97,994 | 1196 |
| | VANDAL | 105,535 | 1,431 | 104,104 | 85,721 |
| | GIGAHORSE | 105,535 | N/A | N/A | 3,310 |
| UncheckedCall | TXSPECTOR | 9,661,593 | 323,772 | 9,337,821 | 178,303 |
| | SECURIFY | 105,535 | 6,494 | 99,041 | 2,380 |
| | VANDAL | 105,535 | 1,151 | 104,384 | 92,379 |
| Suicidal | TXSPECTOR | 9,661,593 | 327,208 | 9,334,385 | 23 |
| | VANDAL | 105,535 | 1,187 | 104,348 | 349 |
| | GIGAHORSE | 105,535 | N/A | N/A | 383 |

Table 3: Comparing reentrancy, uncheckedcall, and suicidal results with other tools. The numbers for TXSPECTOR and SEREUM are transactions numbers, while others represent numbers of contracts. 'N/A' means that the tool is not open-sourced, so we cannot run it and get the timeout/error results.

page [9], a smart contract is considered re-entrant by GIGAHORSE only if "*the contract makes an external call, which can itself re-enter the contract before the first call updates storage*". Therefore, it has a much more restrictive standard than *inconsistent state* defined by TXSPECTOR. For the 3,292 smart contracts flagged by GIGAHORSE but not by TXSPECTOR, the main reason is that there is no transaction showing the *inconsistent state* in our dataset.

**Case study of the DAO contract.** Although TXSPECTOR falls short in detecting some types of re-entrancy attacks (compared to SEREUM), we show it is still effective in detecting the most prominent ones, such as the DAO attack [27]. The DAO contract is where the re-entrancy attack originally happened. It is the No.1 victim of re-entrancy attacks. Over $50 million worth of *Ether* was stolen from DAO [27]. To avoid the loss, the Ethereum community decided to perform a hard fork on the blockchain to return the stolen money, which led to the split of Ethereum blockchain [5].

To inspect the transactions that might have attacked the DAO contract, simply extracting the transactions with DAO as the receiver does not work, since a malicious contract is not the receiver. Instead, we scanned through the raw trace collected by Trace Extractor for each transaction from block 0 to block $7,200,000$, and only kept the transactions whose traces contain the address of the DAO contract. There are 98,914 transactions left after this filtering process.

We applied TXSPECTOR with the re-entrancy Detection Rule on the 98,914 transactions to see how many times the DAO contract has been attacked. We used the same process mentioned before to generate the logic relations and applied the re-entrancy Detection Rule. Since the re-entrancy transactions are more complex than the regular transactions, we increased the Logic Relation timeout threshold to 200*s* and the Detection Rule timeout threshold to 60*s*. After this process, there are still 3,665 transactions that do not have result due to timeout. Among the remaining 95,249 transactions, TXSPECTOR flags 2,108 of them as re-entrancy attacks.

To compare TXSPECTOR with SEREUM, we check how many of these 98,914 transactions are flagged by SEREUM. In particular, SEREUM flagged 2,112 transactions, 2,108 of which are also flagged by TXSPECTOR. That is, all attacks detected by TXSPECTOR are also flagged by SEREUM. There are 4 transactions flagged by SEREUM but not TXSPECTOR. We manually inspected the 4 transactions to see why TXSPECTOR did not flag them. We checked the logic relations of `SSTORE`, `SLOAD` and `JUMPI` to see if there were dependencies that the TXSPECTOR missed. After examination, we confirmed that in these 4 transactions, there were pairs of (`SLOAD`, `SSTORE`) operating the same storage address. However, these pairs have the same `depth`, meaning that they do not meet the condition of *inconsistent state*. Therefore, TXSPECTOR did not flag them as re-entrancy attacks.

## 8.3 Results of UncheckedCall Attacks

Next, we present the detection results of UncheckedCall attacks. After applying the *UncheckedCall* Detection Rules to our dataset, 323,772 transactions (3.4%) did not finish due to timeout. In the 9,337,821 (96.6%) transactions that had results, TXSPECTOR flagged 178,303 transactions as *UncheckedCall*, and there were 1,430 related receiver contracts. 216 of them were open-sourced, and they were related to 28,377 transactions. We manually inspected the 216 smart contracts, and found 213 of them did have the *UncheckedCall* vulnerability. We further investigated why TXSPECTOR mis-flagged the remaining 3 contracts. We found that these 3 contracts have checks on external calls, but in the transactions, the check was not performed due to "out of gas" failure, so TXSPECTOR flagged them. The 3 mis-flagged smart contracts were related to only 4 transactions. It is worth noting that for the remaining 1,214 closed-source smart contracts, we were not able to perform the manually analysis on the contracts. But we did confirmed that at least one `{CALL POP}` was found in their traces, or they did not use at least one of the call return values, which suggest they are indeed attacks according to our detection rules. We also compare the results with those of SECURIFY and VANDAL. The comparison results are summarized in Table 3.

**Comparison with SECURIFY.** We used SECURIFY to detect attacks abusing the *UncheckedCall* vulnerability of the 105,535 receiver smart contracts in our dataset. When analyzing these smart contracts, 2,993 of them (2.8%) did not finish within 60*s*. Moreover, the analysis of another 3,501 (3.3%) smart contracts does not finish due to some run-time errors (*e.g.*, index out of bound), so there is no result for them. After processing, SECURIFY generates results of 99,041 (93.9%) smart contracts, and flagged 2,380 of them as having the *UncheckedCall* vulnerability.

We further looked into the 178,303 transactions flagged by TXSPECTOR. We extracted the receiver contracts of these transactions (1,404 in total), and compared them with the detection result of SECURIFY. There are 1,183 (84.3%) receiver contracts flagged by TXSPECTOR that are also marked by SECURIFY; another 142 (10.1%) are flagged by TXSPECTOR, but SECURIFY does not finish execution due to timeout or run-time errors; there are 79 smart contracts that are flagged by TXSPECTOR, but marked as *Safe* by SECURIFY. In the bytecode-level traces generated by executing these 79 contracts, the call return values are popped by the caller contract, so there is no `JUMPI-CALL` dependency in the traces. We conjecture that the reason why SECURIFY does not flag these contracts might be issues related to the symbolic execution approach it uses.

There are about 1,200 smart contracts only flagged by SECURIFY, but not by TXSPECTOR. We inspected some of them and found that there are *UncheckedCall* vulnerabilities in the smart contracts, but the vulnerable functions are not included in the transactions. Therefore, TXSPECTOR did not detect them.

**Comparison with VANDAL.** When analyzing the 105,535 receiver smart contracts using VANDAL, 1,151 (1.1%) of them did not finish within 60*s*. For the 1,403 smart contracts flagged by TXSPECTOR, 1,367 of them (97.4%) are also marked by VANDAL; the remaining 36 of them are not identified by VANDAL. Through our manual inspection, we found that these 36 smart contracts have the *UncheckedCall* vulnerability. One example is the contract `0x99ECA3`[2]. In this contract, the return value of the `transfer()` function is not checked, which indicates the *UncheckedCall* vulnerability. Therefore, TXSPECTOR is able to identify vulnerable smart contracts that are missed by VANDAL.

VANDAL flagged another 91,012 smart contracts as *UncheckedCall* vulnerability. TXSPECTOR did not detect these smart contracts due to coverage: the vulnerable functions are not included in the transactions of our dataset.

## 8.4 Results of *Suicidal* Attacks

Finally, we present the detection results of *Suicidal* attacks. After applying the *Suicidal* Detection Rules to our dataset, 327,208 transactions (3.4%) did not finish due to timeout. In the 9,334,385 (96.6%) transactions that have results, TXSPECTOR flagged 23 transactions as *Suicidal*. Among them, there were only 18 receiver smart contracts, since 5 of the transactions had a receiver address of `0x0`, meaning that they were killed immediately after creation. We were not able to study the source code of them since their bytecode and storage were erased from the blockchain when they were killed. From the traces of the 23 transactions, we confirmed that there was no permission check on the caller (`msg.sender`). Therefore, TXSPECTOR did not produce false positives. We also compare the results with those of VANDAL and GIGAHORSE. The comparison results are summarized in Table 3.

**Comparison with VANDAL.** We ran VANDAL on all 105,535 smart contracts in our dataset to check how many of them have the *Suicidal* vulnerability. VANDAL marked 349 of them as vulnerable. 13 out of 18 smart contracts flagged by TXSPECTOR were also marked by VANDAL. For the 5 smart contracts not flagged by VANDAL, VANDAL failed to analyze them due to run-time errors and timeout. For the 336 smart contracts flagged by VANDAL only, the main reason is that these smart contracts have the *Suicidal* vulnerability, but they were not killed yet (*i.e.*, function not called). Therefore, TXSPECTOR did not detect them.

**Comparison with GIGAHORSE.** To compare with GIGAHORSE, we retrieved their result of "Accessible selfdestruct" from their website. Among the 105,535 receiver smart contracts in our dataset, GIGAHORSE flagged 383 smart contracts, only one of which was in common with the result of TXSPECTOR. The reason why GIGAHORSE did not flag the other 17 contracts is that their bytecode were missing after being killed when GIGAHORSE was deployed, so GIGAHORSE cannot analyze them. For the smart contracts flagged by GIGAHORSE only, we found that they have a much relaxed criterion: as long as the `SELFDESTRUCT` is reachable from public entry point [9], it would be flagged as True, even if there are checks. However, TXSPECTOR only detects *Suicidal* vulnerability that has no check at all, which is stricter than GIGAHORSE.

## 8.5 Comparison with Other Tools

In addition to the three Datalog-based tools, we also performed a comparison with three other static analysis tools: MYTHRIL [7], OYENTE [31] and MAIAN [35]. For MYTHRIL, we compared it with TXSPECTOR on all three vulnerabilities; we compared OYENTE on *re-entrancy* and MAIAN on *suicidal*, respectively. The results are summarized and presented in Table 4. To summarize, the detection results varied a lot for different tools. The main reason is that there are no *golden rules* for detecting these vulnerabilities and different tools use different detection rules. In addition to making our source code open, we have also released our comparison results so that others in the community can use the data for their research.

## 8.6 Timeout Analysis

**Timeout due to generating logic relations.** When generating logic relations, 1,082 transactions failed to finish within 60s. We manually inspected these transactions to understand

[2] `0x99ECA38B58cEEaf0FeD5351DF21D5B4C55995314`

| Tool | # Reentrancy | # UncheckedCall | # Suicidal |
|---|---|---|---|
| TXSPECTOR | 30 | 1,430 | 18 |
| SECURIFY | 1,196 (0) | 2,380 (1,183) | N/A |
| GIGAHORSE | 3,310 (18) | N/A | 383 (1) |
| VANDAL | 85,721 (27) | 92,379 (1,367) | 349 (13) |
| MAIAN | N/A | N/A | 21 (7) |
| OYENTE | 9,556 (6) | N/A | N/A |
| MYTHRIL | 19,854 (6) | 52 (31) | 1(1) |

Table 4: Comparison with static analysis tools. $a(b)$ means the tool flags $a$ contracts, and $b$ of them are in common with the result of TXSPECTOR.

the reasons of the timeout. We found that most of these transactions got stuck in the Logic Relation Builder (§6), which converts trace-based EFGs to IR and performs arithmetic operations with real values. The arithmetic operations may at times be too complex to compute on-the-fly (*e.g.*, $exp(a,b)$), which is the main reason of timeout due to logic relation generation. One example is transaction `0xf9de18` [16], which has 6,945 OPCODEs. Also, it has an *exp* operation with a large number as the exponent, which causes the timeout.

**Timeout due to applying Detection Rules.** We analyzed the transactions that exceed the timeout threshold when applying *UncheckedCall* Detection Rules. We found that most of the transactions got stuck when finding the dependencies between call return values and `JUMPI`. Assume that the number of `JUMPI` is $m$ and the number of `CALL` is $n$ in a transaction trace, there will be $m*n$ (`CALL`, `JUMPI`) pairs. For each pair, TXSPECTOR tries to check whether the call return value is used by the `JUMPI` by finding the dependencies between them, possibly through many intermediate variables. When $m*n$ is large and the trace is long, it would take a lot of time validating the dependencies of all $m*n$ pairs. One example is transaction `0xb513f5` [15], which has 11,664 `JUMPI` and 299 `CALL`. For about 3.5 million (`CALL`, `JUMPI`) pairs, TXSPECTOR needs to go over all potential intermediate variables to confirm whether there is a dependency, which is unbearable.

**Optimizations.** For optimizing the logic relation generation, we can fetch these intermediate results from the Ethereum node, since they should be present during transaction execution. To speed up the Detection Rule application process, we can add stricter pruning rules to filter out pairs that are impossible to have dependencies, before trying to find them. Also, we may add some helper functions to store dependencies of certain nodes so that it does not have to be re-computed every time. We leave these optimizations as our future work.

# 9 Application

In this section, we demonstrate how TXSPECTOR can be used to perform forensic analysis of attacks against Ethereum.

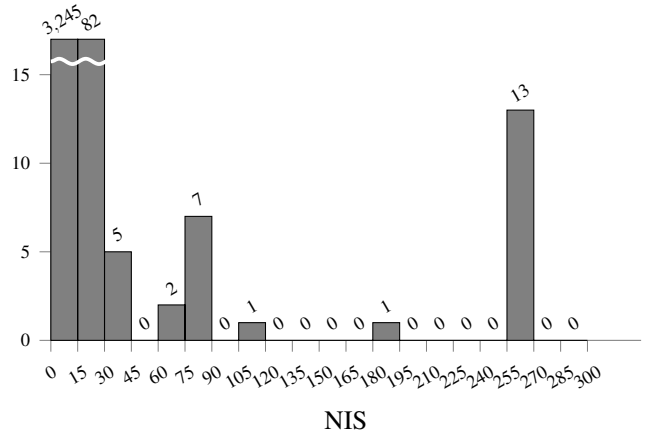


Figure 9: Distribution of NIS in transactions flagged as re-entrancy attacks. Note that there is one transaction that has 575 NIS not presented in this figure.

## 9.1 Forensic Analysis of Re-entrancy

First, we focused on the 3,327 transactions flagged by TXSPECTOR as re-entrancy attacks. We inspected these transactions closely to study the following aspects:

**The number of *inconsistent state* (NIS).** We defined NIS in a transaction as the number of *inconsistent state* reported in the query result by TXSPECTOR, which is the number of different (`SLOAD`, `SSTORE`) pairs that operate the same storage address in a transaction. The distribution of the NIS in the 3,327 flagged transactions is shown in Figure 9. In this figure, the X-Axis shows the number of NIS and the Y-Axis indicates the number of transactions whose NIS falls in the corresponding range. We can see that there are 3,245 transactions with an NIS smaller than 15. Over 15 transactions have more than 100 *inconsistent state*; there is one transaction with 575 NIS.

**Victim smart contracts.** After studying the real-world transactions that are involved in re-entrancy attacks, we found that the typical attack workflow is as follows: (*i*) An externally owned account ($A$) calls a function in a smart contract ($B$). Both $A$ and $B$ are controlled by the attacker; (*ii*) $B$ calls another smart contract ($C$), which is the victim; (*iii*) $C$ calls the fallback function of $B$. In this fallback function, $B$ re-enters $C$, and the attack repeats until an exception happens.

In this workflow, $B$ is the malicious smart contract, and $C$ is the victim smart contract. In Ethereum, the sender (*from* address) of the attack transaction is $A$, and the receiver (*to* address) is $B$. Therefore, the receiver of the transaction is not the victim; the actual victim ($C$) is hidden in the bytecode or storage of $B$. This finding is slightly counter-intuitive.

To investigate deeply of our findings, we constructed a new Detection Rule to extract the address of the contract who is the

| Address | NIS Count |
|---|---|
| 0xdf18880a02c7f3eb4f40fdf515fce31c1cb7ef66 | 4,803 |
| 0x1806b3527c18fb532c46405f6f014c1f381b499a | 3,815 |
| 0xd7a14019aeeba25e676a1b596bb19b6f37db74d2 | 2,839 |
| 0x533bafa16aa76218ec4a365ad71bf8816cf21bbb | 675 |
| 0x431d77f50803d31b090e86740b1d5848af54fad0 | 582 |

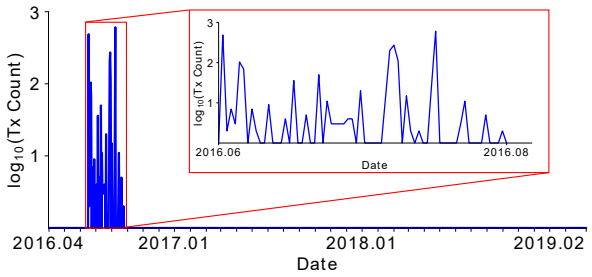


Figure 10: Distribution of re-entrancy attacks on DAO.

owner of the storage address in an *inconsistent state*. For a single transaction, if it contains bytecode from multiple smart contracts that have *inconsistent state*, the Detection Rule reports the addresses of these smart contracts, as well as their NIS numbers, respectively. After the query, Attack Detector reports 318 unique victim addresses, the top 5 of which are shown in Table 5. The top victim smart contract accounts for 4,803 NIS. The top 2 smart contracts combined contribute to over 8,600 NIS counts, which is more than half of the total NIS counts of all 318 victim smart contracts together.

**Case Study – The DAO contract.** A well-known re-entrancy attack is the DAO attack. We therefore performed a case study on the DAO smart contract, focusing on the time of the detected attacks and their NIS numbers. As mentioned in §8.2, there are 98,914 transactions related to DAO. TxSpector flags 2,108 of them as re-entrancy attacks. The distribution of these 2,108 transactions is shown in Figure 10. From this figure, it is clear that most of the re-entrancy attacks on DAO happened in summer 2016, which is consistent with the news report on the infamous DAO attack [27]. We further studied the NIS of the re-entrancy transactions targeting the DAO contract. The distribution of NIS is shown in Figure 11. Not surprisingly, re-entrancy transactions on DAO have much larger NIS counts. We can see that there are more than 1,700 transactions with an NIS larger than 100; Over 850 transactions have more than 500 *inconsistent state*; there are even 512 transactions with an NIS larger than 1,700.

## 9.2 Forensics Analysis of UncheckedCall

TxSpector flagged 178,229 transactions as attacks exploiting the *UncheckedCall* vulnerability, and they have 1,404 unique receiver addresses and 4,125 unique caller addresses.

**Receiver Address.** We listed the top 5 receiver addresses of *UncheckedCall* transactions in Table 6. The top receiver smart contracts account for more than 30,000 transactions. The top

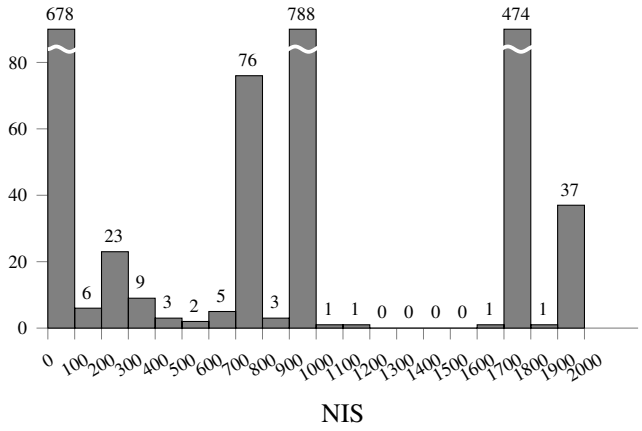


Figure 11: NIS in re-entrancy transactions on DAO.

3 receiver smart contracts combined contributed to about 50% of all *UncheckedCall* transactions. The No.3 smart contract belongs to *HybridExchange* [11], which is a crypto wallet as well as an exchange platform.

**Caller Address.** We listed the top 5 caller addresses of *UncheckedCall* transactions Table 6. The top caller was responsible for sending over 24,000 *UncheckedCall* transactions. The No.2 to No.5 callers each sent over 7,500 *UncheckedCall* transactions.

## 9.3 Forensic Analysis of Suicidal

TxSpector flagged 23 transactions as *Suicidal*. We first investigated the reasons behind the 23 *Suicidal* transactions. Recall that the main requirement for *Suicidal* is *no permission check before* `SELFDESTRUCT`. After our investigation, we found that the reasons can be categorized into two classes:

- **No permission check at all:** There is no check at all in the transaction. 20 transactions fall into these category, meaning that they can be killed by anyone, as expected.
- **Mistakes in checks:** There are checks in the transaction, but it does not check the *msg.sender*. We find that there

| Category | Address | Tx Count |
|---|---|---|
| Top 5 Receiver Addresses | 0x827727b4c3f75ea6eb6bd2cc256de40db2b13665 | 30,705 |
| | 0x896b516eb300e61cfc96ee1de4b297374e7b70ed | 28,912 |
| | 0x2cb4b49c0d6e9db2164d94ce48853bf77c4d883e | 24,254 |
| | 0x0000002c2155eb1aaa8809e93f88873ddcf40c55 | 9,102 |
| | 0x3d374d549f78503f3252fa18cc02237da008c9f7 | 8,524 |
| Top 5 Caller Addresses | 0x49497a4d914ae91d34ce80030fe620687bf333fd | 24,254 |
| | 0x17528a9314b090a13a97b4f167d7d525625c398d | 7,735 |
| | 0xe8a576d484c10bed29aed74d16d6958aa05f94aa | 7,703 |
| | 0x62460a5567d2823781604dc938e0eaf073d24d9d | 7,662 |
| | 0x682ed78859e2235e03535e11d2396e1e200bf0d4 | 7,605 |

Table 6: Top 5 Receiver addresses and Caller addresses in *UncheckedCall* transactions.

| Beneficiary Address | Tx Count |
|---|---|
| 0x3a91b432b27eb9a805c9fd32d9f5517e9dd42aa4 | 3 |
| 0x6e226310db63ac3701f657bcc62c153c1aaa3004 | 2 |
| 0x15202d3d183708649451878f50982d5c1bb4d01b | 2 |

Table 7: Common beneficiary addresses.

are 3 transactions containing checks, but they only check the *origin*, rather than the *msg.sender*, which is actually a *Misuse-of-origin* vulnerability. As a result, they can be killed by arbitrary caller.

Next, we inspected the flagged 23 transactions to study the caller address and the beneficiaries of the attacks:

- **Caller Address.** We checked whether these transactions were triggered by the same caller by clustering them based on the caller address. There were 4 sets of transactions having the same caller address, which have 3, 2, 2, 2 transactions in each set, respectively. We further checked the bytecode of the smart contracts in each set, and confirmed that they were actually identical. Contracts in each set are created by the same creator.
- **Beneficiaries.** When a smart contract (A) executes `SELFDESTRUCT`, it needs to specify an address of another account (B). The remaining Ether of A will be transferred into B. Therefore, we call B the *beneficiary*. We further checked the beneficiary address of these 23 transactions, and the common beneficiaries are shown in Table 7. There were 3 sets of transactions having the same beneficiary, respectively; the top one is the beneficiary of 3 *Suicidal* transactions.

## 10 Discussion

**Time cost.** As shown in §8.1, it takes 1.03*s* on average to generate the logic relations for one transaction. Considering the amount of transactions in Ethereum, processing them in real-time would be very challenging. TXSPECTOR is designed as a forensic analysis framework on transactions, but not intended to be used as a real-time attack detection tool for Ethereum. Nevertheless, there are several ways to improve the performance of TXSPECTOR when generating logic relations. For example, multi-threading can be applied to generate logic relations of multiple transactions in parallel, since there is no dependency among the transactions.

**Storage cost.** It takes a lot of space to store the Logic Relation DB. To save space, TXSPECTOR can take measures to shrink the size of the Logic Relation DB. For example, standard serialization or compression libraries (*e.g.*, *gzip*) can be used when generating the logic relations. Moreover, TXSPECTOR can choose a subset of OPCODEs and only generate logic relations for these OPCODEs, instead of all of them, if the OPCODEs of interest are known before going through the Logic Relation Builder.

**Transaction vs. bytecode.** The benefit of studying transactions is that transactions contain information of how smart contracts interact with each other. Nevertheless, the bytecode-level trace of a transaction only contains partial information of smart contracts; it only involves the functions that are invoked during this transaction. If a function in a smart contract is never invoked by others, there is no transaction associated with it. In this case, transactions cannot reveal any vulnerabilities related to this function of the smart contract. Nevertheless, if a smart contract is never involved in any transaction, the vulnerabilities are not exploited, either. Therefore, for forensic analysis, analyzing transactions is more meaningful than studying smart contract bytecode.

**Reactive approach vs. proactive approach.** As an attack detection and forensic analysis tool, TXSPECTOR examines transactions, which is reactive in nature, meaning that attacks can only be detected *after* they have occurred on the Ethereum blockchain. Unlike the proactive approaches (*e.g.*, static analysis) that detect vulnerabilities in smart contracts which may never be triggered, however, studying transactions can reveal *true* attacks happened in the past, and learn from them in a forensic perspective. On the other hand, static analysis tools complement TXSPECTOR since TXSPECTOR can only see parts of the smart contract bytecode. After TXSPECTOR uncovers an attack from a transaction, static analysis tools can be used to study the victim smart contract to identify other potential attack surfaces, as well as the attacker smart contract to learn about the attack mechanisms.

**Efforts needed to design new rules.** TXSPECTOR can only be used to perform forensic analysis on *known* attacks/vulnerabilities. In order to come up with the Detection Rules, the user needs to have some knowledge of the attacks/vulnerabilities she wants to detect, as well as the basic understanding of constructing the Detection Rules. In the open-source release of TXSPECTOR, we have provided rules of existing vulnerabilities for the users to choose from, so that they do not have to reinvent the wheel. Moreover, to minimize the effort that the user needs to put to develop a customized Detection Rule, we have also provided a list of APIs, as well as documentations/READMEs to help the user get on-board.

**Other applications.** In this paper, we show that TXSPECTOR can be used to detect 6 different kinds of attacks. However, the applications of TXSPECTOR are beyond detecting attacks and vulnerabilities; it can be used to perform forensic analysis on many other aspects from the transactions. For example, as shown in §9.1, TXSPECTOR can be used to check whether a specific address is involved in a transaction, and, if so, perform certain analysis on the transaction. Also, it can be used to retrieve certain intermediate results to learn about transaction failure reasons.

## 11 Conclusion

We have presented TXSPECTOR, the *first* generic, open source, and logic-driven framework for studying Ethereum transactions at the bytecode level. TXSPECTOR supports customized Detection Rules defined by the users to detect Ethereum attacks. We present the design and implementation of TXSPECTOR, and demonstrate the construction of Detection Rules for detecting attacks in transactions. Our evaluation suggests that TXSPECTOR is effective. We also demonstrate how to use TXSPECTOR to perform forensic analysis on transactions.

## Acknowledgement

We thank the anonymous reviewers for their valuable comments. We also thank our shepherd, Thorsten Holz, for helping us improve our paper. This work is supported in part by the NSF grants 1718084, 1750809, 1834213 and 1834215.

## References

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

[2] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2016.

[3] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[4] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for ethereum to defend against underpriced dos attacks," in *International Conference on Information Security Practice and Experience*. Springer, 2017.

[5] Coindesk, "Ethereum's two ethereums explained," https://www.coindesk.com/ethereum-classic-explained-blockchain.

[6] ConsenSys, "Consensys/mythril: unchecked call return values," https://github.com/ConsenSys/mythril/blob/develop/mythril/analysis/modules/unchecked_retval.py.

[7] ——, "Mythril classic," https://github.com/ConsenSys/mythril-classic.

[8] Dedaub, "Ethereum contract library," https://contract-library.com.

[9] ——, "Faq | ethereum contract library," https://contract-library.com/faq.

[10] E. Documentation, "Account types, gas, and transactions," http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html.

[11] Eidoo, "Hybrid exchange - eidoo," https://eidoo.io/hybrid-crypto-exchange.

[12] Eth-sri, "Gas-dependent reentrancy - securify," https://github.com/eth-sri/securify/blob/master/src/main/java/ch/securify/patterns/DAO.java.

[13] ——, "Reentrancy with constant gas - securify," https://github.com/eth-sri/securify/blob/master/src/main/java/ch/securify/patterns/DAOConstantGas.java.

[14] ——, "Securify: Security scanner for ethereum smart contracts)," https://github.com/eth-sri/securify.

[15] Etherscan, "Detection rules timeout example," https://etherscan.io/tx/0xb513f563987c842e1cbe65652de602069fbc5ba2-42eef77904497b69078f807b.

[16] ——, "Logic relations timeout example," https://etherscan.io/tx/0xf9de1870affa8f6a760a2f330e4ede41b17eb10e98cc1af6e27393c78-a613a17.

[17] ——, "Re-entrancy false positive example," https://etherscan.io/tx/0xd324962339c04ad16138a2b9a9732063c35221bf538b55e275dee6-a7cba78f8d.

[18] ethervm.io, "Ethereum virtual machine opcodes," https://ethervm.io.

[19] ——, "Online solidity decompiler," https://ethervm.io/decompile.

[20] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019.

[21] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[22] E. Github, "Solidity, the contract-oriented programming language," https://github.com/ethereum/solidity.

[23] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *International Conference on Software Engineering (ICSE)*, 2019.

[24] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, 2017.

[25] hackingdistributed, "Scanning live ethereum contracts for the "unchecked-send" bug," http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/.

[26] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu, "Kevm: A complete semantics of the ethereum virtual machine (2017)," *White paper*, 2017.

[27] B. Insider, "Digital currency ethereum is cratering because of a $50 million hack," https://www.businessinsider.com/dao-hacked-ethereum-crashing-in-value-tens-of-millions-allegedly-stolen-2016-6?r=UK.

[28] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*. Springer, 2016.

[29] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[30] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium*, 2018.

[31] 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.

[32] Microsoft, "Virtual machine series | microsoft azure," https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/.

[33] MongoDB, "Mongodb: The most popular database for modern apps," https://www.mongodb.com/.

[34] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[35] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018.

[36] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A formal verification tool for ethereum vm bytecode," in *Proceedings of the 26th*

*ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018.

[37] D. Perez and B. Livshits, "Smart contract vulnerabilities: Does anyone care?" *arXiv preprint arXiv:1902.06710*, 2019.

[38] M. Rodler, W. Li, G. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proceedings of the 26th Network and Distributed System Security Symposium*, 2019.

[39] smartdec, "Smartcheck - a static tool to detects vulnerabilities in solidity programs," https://github.com/smartdec/smartcheck.

[40] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss *et al.*, "Dependent types and multi-monadic effects in f," in *ACM SIGPLAN Notices*. ACM, 2016.

[41] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2018.

[42] T. N. Y. Times, "A hacking of more than $50 million dashes hopes in the world of virtual currency," https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html.

[43] TrailOfBits, "Manticore: Symbolic execution tool," https://github.com/trailofbits/manticore.

[44] 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.

[45] usyd blockchain, "Vandal: Static program analysis framework for ethereum smart contract bytecode)," https://github.com/usyd-blockchain/vandal.

[46] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[47] Zeppelin, "The parity wallet hack explained - zeppelin blog," https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7.

# A Other Detection Rules

## A.1 Timestamp Dependence

**Description.** When a smart contract's control flow depends on the states of the block or the transaction, it can be abused by a miner who may manipulate these states. For example, a contract may use the `TIMESTAMP` OPCODE to obtain the timestamp of the current block, and perform certain actions based on the result. However, the `TIMESTAMP` may be manipulated by the miners. If a smart contract contains a conditional jump which contains such OPCODEs in the condition, it has the *Timestamp Dependence* vulnerability.

**Requirements.** We adapt the detection criteria of another bytecode analysis tool, SMARTCHECK [39, 41], to define the requirements of *Timestamp Dependence* attacks in transactions. There are two requirements: (*i*) **`TIMESTAMP`:** There is at least one `TIMESTAMP` OPCODE in the transaction. (*ii*) **`TIMESTAMP-JUMPI` dependency:** there is a `JUMPI` which depends on the result of the `TIMESTAMP` OPCODE. If a transaction satisfies the requirements, it means that the current timestamp is used in a control-flow decision, which leads to the *Timestamp Dependence* attacks.

```
1  TimestampDependence(args):-
2    % TIMESTAMP-JUMPI dependency
3    op_TIMESTAMP(_, tsVal, tsIdx, 1, _),
4    jumpiDep(jumpiIdx, 1, tsIdx, tsVal).
```

Figure 12: *Timestamp Dependence* Detection Rules.

**Detection Rules.** The goal of the Detection Rules is to detect the `TIMESTAMP-JUMPI` dependency. Similar to the Re-entrancy Detection Rules shown in Figure 5, we define the Detection Rules for detecting *Timestamp Dependence* attacks in Figure 12. The Detection Rules extract all `TIMESTAMP` and `JUMPI` in the transaction, and check whether there is a `TIMESTAMP-JUMPI` dependency. The `depth` of them is set to 1 to capture the attack of the receiver smart contract only.

## A.2 Misuse of Origin

**Description.** Let `tx.origin` denote the original sender of a transaction and `msg.sender` denote the immediate sender. The `tx.origin` (`ORIGIN` OPCODE) returns the address of the first message sender of the transaction, rather than the caller of current function, *i.e.*, `msg.sender` (`CALLER` OPCODE). When a smart contract mistakenly use the `ORIGIN` to check its caller, it contains the *Misuse-of-origin* vulnerability, which can be exploited by a malicious contract that relay transactions.

**Requirements.** To check *Misuse-of-origin* attacks, TXSPECTOR needs to examine the following two requirements: (*i*) **`ORIGIN`:** The transaction contains a `ORIGIN` OPCODE. (*ii*) **`ORIGIN-JUMPI` dependency or `ORIGIN-SSTORE` dependency:** There is a conditional jump or a storage write that depends on the result of `ORIGIN`. Note that if it is a conditional jump, it should not come from a comparison between *tx.origin* and `msg.sender` (*e.g.*, `tx.origin` == `msg.sender`), which is a legitimate usage of `tx.origin`.

When a conditional jump or a storage write depending on the `tx.origin`, and it is not a comparison between `tx.origin` and `msg.sender`, it means that the `ORIGIN` is misused, thus a *Misuse-of-origin* attack.

**Detection Rules.** The Detection Rules for detecting *Misuse-of-origin* are shown in Figure 13. The Detection Rules first extract all the `ORIGIN` OPCODEs in the transaction. Then the Detection Rules check the two conditions, respectively: For the `ORIGIN-JUMPI` dependency, they find all `JUMPI`s that depend on `ORIGIN` (line 5), and remove those comparisons between `tx.origin` and `msg.sender` via the **cmpOrigin** Detection Rule (line 6); For the `ORIGIN-SSTORE` dependency, they find all `SSTORE`s (line 9) and check whether any of them depends

```
1  MisuseOfOrigin(args) :-
2    op_ORIGIN(_, originRet, originIdx, 1, _),
3
4    % ORIGIN-JUMPI dependency
5    ((jumpiDep(jumpiIdx, 1, callIdx, originRet),
6    !cmpOrigin(jumpiIdx));
7
8    % ORIGIN-SSTORE dependency
9    (op_SSTORE(_, sstoreAddr, _, sstoreIdx, 1, _),
10   depends(sstoreAddr, originRet))).
```

Figure 13: *Misuse-of-origin* Detection Rules.

```
1  FailedSend(args) :-
2    op_CALL(_, ether, _, callRet, callIdx, 1, _),
3    !value(ether, "0x0"), value(callRet, "0x0"),
4    jumpiDep(jumpiIdx, 1, callIdx, callRet),
5    op_REVERT(_, _, _, revertIdx, 1, _).
```

Figure 14: The Detection Rules for detecting *FailedSend*.

on ORIGIN (line 10). The two conditions are concatenated with a ';', which means the *OR* operator.

## A.3 FailedSend

**Description.** The *FailedSend* vulnerability is similar to the *UncheckedCall* vulnerability, but the smart contract throws an exception when the call fails. This may cause problems as well. For example, when smart contract A sends money to smart contract B, the fallback function of B will be called. If B is malicious, it can do something to make the money-transfer operation fail, *e.g.*, put a long sequence of OPCODEs in its fallback function to make it cost more than the gas limit (2300 *wei* by default). As a result, the fallback function can never succeed and the transaction will be reverted.

**Requirements.** To check the *FailedSend* attacks, there are 3 requirements: (*i*) **Failed *Send()*:** The transaction should contain a CALL-related OPCODE, and the *Ether* to be transferred is greater than 0, indicating a *Send()* operation. Moreover, the result of this CALL should be *False*, meaning that it fails. (*ii*) **Checked call return value:** There is a JUMPI that depends on the return value of the CALL. (*iii*) **REVERT:** The caller reverts the transaction.

If a transaction satisfies all the requirements, it means that the caller failed to send *Ether* to the callee. Also, after checking the return value (via JUMPI), the transaction is reverted by the caller. Therefore, they are necessary and sufficient conditions of a *FailedSend* attack.

**Detection Rules.** The Detection Rules for detecting the *FailedSend* attacks is shown in Figure 14, which are very similar to the Detection Rules for *UncheckedCall*. First, the *FailedSend* Detection Rules extract all external calls in a transaction (line 2), then check the *Ether* amount (line 3) and the return value (line 4). After that, The Detection Rules try to find whether there is a REVERT after the JUMPI (line 5-6). Note that the depth of both CALL and JUMPI in our Detection Rules is set to 1 as well, the same as in *UncheckedCall*. Since the REVERT will always be the last OPCODE in the transaction, there is no need to compare revertIdx and jumpiIdx.

## A.4 Gas-dependent Reentrancy and Reentrancy with Constant Gas

The Detection Rules for Gas-dependent Reentrancy [12] and Reentrancy with Constant Gas [13] defined by SECURIFY [44] are presented in Figure 15. There are three requirements for them: (*i*) there is an Ether transfer; (*ii*) there is a state change (SSTORE) after the call returns; (*iii*) the value of Ether transferred depends on the storage variable. By utilizing the above conditions, we can find those reentrancy attack transactions as defined in SECURIFY. If the gas of the *Send()* is a constant, then the attack is related to *Reentrancy with Constant Gas*; otherwise, the attack is related to *Gas-dependent Reentrancy*.

## A.5 Generic Detection Rules

Generic rules, instead of rules targeting specific attacks, can also be expressed in TXSPECTOR, as long as the information needed is all present in the transaction traces. Although we only present rules for detecting specific attacks in this paper, generic rules can also be defined. For example, in Figure 16, we show Detection Rules to detect transactions that involve at least *n* smart contracts, and transactions that run out of gas.

```
1  Requirements(gas) :-
2    op_CALL(_, ether, gas, callRet, callIdx, callDepth, _),
3    !value(ether, "0x0"), value(callRet, "0x1"),
4    op_SSTORE(_, sstoreAddr, _, sstoreIdx, sstoreDepth, _),
5    sstoreDepth = callDepth, sstoreIdx > callIdx,
6    depends(ether, sstoreAddr).
7
8  ReentrancyGasConst(args) :-
9    Requirements(gas), isConst(gas).
10
11 ReentrancyGasDep(args) :-
12   Requirements(gas), !isConst(gas).
```

Figure 15: The Detection Rules for detecting reentrancy with constant gas and gas-dependent reentrancy.

```
1  CallAddress(callAddr) :-
2    op_CALL(callAddr, _, _, _, _, _, _).
3
4  InvolveNContracts(args) :-
5    total = count:{CallAddress(callAddr)},
6    % n is a constant
7    total >= n.
8
9  OutOfGas(args) :-
10   % Last returns the max idx and the related OP
11   Last(lastIdx,lastOp),
12   lastOp != "STOP", lastOp != "SELFDESTRUCT",
13   lastOp != "RETURN", lastOp != "REVERT",
14
15   % CurrentGas returns the remaining gas
16   CurrentGas(lastIdx, gas),
17   value(gas, "0x0").
```

Figure 16: Examples of generic Detection Rules.