Detect Abnormal Behaviours in Ethereum Smart Contracts Using Attack Vectors

Quoc-Bao Nguyen[1], Anh-Quynh Nguyen[2], Van-Hoa Nguyen[3], Thanh Nguyen-Le[3], and Khuong Nguyen-An[1(✉)]

[1] University of Technology (HCMUT), VNU-HCM, Ho Chi Minh City, Vietnam
{1510180,nakhuong}@hcmut.edu.vn
[2] Nanyang Technological University, Singapore, Singapore
aqnguyen@ntu.edu.sg
[3] Verichains Lab, Ho Chi Minh City, Vietnam
{vanhoa,thanh}@verichains.io

Abstract. Blockchain has gradually been popularized by its transparency, fairness, and democracy. This technology has opened the door to the development of Ethereum, a blockchain platform with smart contracts that can hold and automatically transfer tokens. Like a legacy computer program, smart contracts are vulnerable to security bugs. In recent years, many successful attacks on Ethereum network have been recorded, cost victims millions of dollars. In this paper, we classify attack vectors of Ethereum smart contracts, then propose some behaviour-based methods to detect them. To realize the ideas, we implement Abbe, a tool that can not only discover known attacks but also detect zero-day vulnerabilities.

Keywords: Smart contract · Security · Ethereum · Blockchain

1 Introduction

The term *smart contract* was conceived by Szabo [24] in 1994, which simply describes a computer program facilitating the terms and conditions of a real-world contract. These negotiations are presented as 'if-else' statements. For example, "if A transfers money to B, A may gain access to B's apartment". Different from normal contracts, smart contracts automatically enforce these negotiations once the predefined conditions are satisfied, without the interference of an authorized notary or a trusted third-party. Although this concept has existed since the early of 90s, due to a missing prerequisite, namely, a decentralized platform, smart contracts could not come into use until the birth of blockchain technology.

Since Nakamoto [18] introduced the concept of blockchain in 2008, a large amount of attention has been paid to this prominent technology. At this time, Nakamoto represented his idea through an implementation version named Bitcoin. The blockchain technology is claimed to have many interesting properties,

T. K. Dang et al. (Eds.): FDSE 2019, LNCS 11814, pp. 485–505, 2019.
https://doi.org/10.1007/978-3-030-35653-8_32

including *transparency*, *fairness*, and *democracy*. These properties are manifested whilst all member nodes in the network hold a replica of the blockchain shared-data. The data is publicly-verifiable, tamper-resistant even in the absence of a trusted party. This is because all nodes have equal rights to commit new updates into the shared-data of blockchain. In order to guarantee the network synchronization, every new update must be processed through a cryptographic scheme which ensures all new records are verified by all nodes and securely appended into the database of these nodes. In 2014, in the a improvement proposal of Bitcoin, a new op-code was added to describe a specific time in the future at which the transaction will be executed. There remains the fact that this enhancement is not enough to make up the features of a smart contract. In 2013, Buterin *et al.* [8] first combined the concept of smart contract and the blockchain technology and successfully implemented into a platform named Ethereum. Among current blockchain platforms, Ethereum, whose capitalization has reached 34 million dollars in June 2019[1], stands out to be the most prominent framework supporting smart contracts.

Due to being developed and running on top of blockchain infrastructure, smart contracts inherit certain interesting properties. The bytecode of smart contracts is immutable i.e., cannot be tampered once deployed. Moreover, the results returned after smart contract execution are irreversible and permanently recorded inside the blockchain database. However, smart contracts basically are computer programs which may contain vulnerabilities due to either mistakes of developers or the instinct of programming language. Recall that the most interesting property of smart contracts is the ability to hold *ether*, which creates economic incentive to be attacked. Many security vulnerabilities have been discovered by systematic exposition [2], practical development experience [10] and analysis of Ethereum smart contracts [14]. Some of these vulnerabilities actually *have been* exploited in real world attacks leading to *hundred millions dollars* drained [1,11,20,23].

To address these issues, many security research teams [14,17,21,26] have conducted solutions for auditing smart contracts' source code before deploying them onto the blockchain. However, this approach only deals with security issues in the development stage, which is before the execution stage in a life-cycle of a product. It is demanding to identify whether the execution of smart contracts contains any security issues. A solution addressing this problem can be helpful for smart contracts' developer team as they can detect abnormal behaviours, or worst, recognize money loss in their contracts and make quick responses to those incidents. Moreover, this solution can be used by blockchain developer to improve their infrastructure regarding to occurred issues. However, up to now, there is no related work tackling the above-mentioned problem.

Indeed, the smart contracts cannot start to execute themselves but have to be triggered by several external entities such as other contracts or, especially, by users. The appearance of security issues is now mostly from the transactions that users send to the smart contracts. This paper proposes several methods

[1] https://coinmarketcap.com/currencies/ethereum/.

to discover abnormal behaviours in Ethereum smart contracts of Ethereum by detecting their attack vectors. We built a tool named ABBE to demonstrate the methods.

Our contributions in this paper are threefold:

- Our work collects known vulnerabilities from many scattered sources and systematically classified them.
- We then define an attack vector for each vulnerability. Afterwards, we record all tracks that are left by each attack vectors and classified them according to some properties.
- Since each property in these tracks category requires different preliminaries to be detected, we also implement several modules to handle them. A tool named ABBE, is implemented and can detect abnormal executions in all transactions of Ethereum smart contracts.

2 Backgrounds of Ethereum Smart Contracts

Ethereum is a transaction-based state machine in which smart contracts execute as regards the intentions of developers. Ethereum has its own virtual machine, named Ethereum Virtual Machine (EVM), to execute smart contract code.

```
1  contract BobCompany {
2      mapping (address => uint) stock;
3      uint stockPrice;
4      constructor () {
5          stock[0x0000000000000000000000000000000deadbeef] = 100
6      };
7      function buyStock (address _from, address _to, uint _amount) payable
           {
8          if (stock[_from] == 0 || msg.value == 0) throw;
9          if (stock[_from] > _amount && _amount*stockPrice < msg.value) {
10             stock[_from] -= _amount;
11             stock[_to] += _amount;
12         }
13     }
14     function sellStock(address _from, uint _amount) {
15         if (stock[_from] > _amount) {
16             stock[_from] -= _amount;
17             msg.sender.transfer(_amount*stockPrice);
18         }
19     }
20 }
```

Listing 1. An example of smart contract

Programming. At first, smart contracts are programmed in high-level programming languages, e.g. **Solidity** (a likewise of C and Javascript), **Vyper** (a contract-oriented language), **LLL** (a low-level language similar to Lisp), etc. A simple smart contract written in Solidity is illustrated in Listing 1 which features buying stock of Bob company. In this contract, Bob owns initially 100 percent of his company stock which defined in `constructor` function (line `5`). This constructor executes exactly once during the **deployment** of contract.

Deployment. In order to be run on-top of EVM, the source code is compiled into EVM bytecode [27]. Afterwards, an address is determined for this new contract. Information of the address along with its corresponding bytecode is

stored by all nodes in blockchain network assuring these data could not be altered after deploying.

Execution. Assume that, Alice—a businesswoman—is willing to buy a couple of stock from Bob's company, she actually sends an appropriate amount of *ethers*[2] via a *transaction* to the contract. Afterwards, the contract verifies some pre-defined conditions to proceed her demand. For example, if Bob held no stock or Alice did not send any *ether* (line `8`), the contract could raise an exception to revert all temporary changes, return Alice's ether, etc., but all fees are consumed. Otherwise, if all conditions were satisfied as shown in line `9`, the contract would make changes in Alice and Bob's balances, respectively. Moreover, if Alice calls `sellStock` function (line `14`) to sell a couple of her stocks, the contract will essentially create a *call*, a.k.a. *internal transaction*, to finish her order (line `19`). Note that, a contract is able to send ethers to another normal account or invoke functions of other smart contracts. It is important to be aware that each transaction may interact with contract to invoke a function and requires fees (named *gas*[3] in Ethereum) to be processed by *all* nodes in the blockchain. Due to this reason, smart contracts are only suitable for low computational effort tasks such as signature verification, money transfer. The restriction placed by gas is for preventing the whole Ethereum network from being abused.

3 Taxonomy of Vulnerabilities

3.1 Reentrancy

In terms of transaction processing, *atomicity* and *sequentiality* are two among four essential properties of a transaction. Those properties are represented as a non-recursive function cannot be invoked until the function execution state ends up in a terminated state. However, *fallback* and *call* functions may break those properties of contracts, since they allow an adversary to continuously recall, or to *re-entrant*, before the termination of the transaction. This vulnerability is illustrated in a simple version contract (see Listing 2) of the historical attack of the Decentralized Autonomous Organization (DAO) [11] in 2016, which caused the loss of $60M dollars in total.

```
contract Alice {
    bool sent = false;
    function send(address c){
        if (!sent) {
            c.call.value(1)();
            sent = true;
        }
    }
}
```

```
contract Fraudster {
    function () payable {
        Alice(msg.sender).send(this);
    }
}
```

Listing 2. Contract contains reentrancy

[2] Name of the cryptocurrency used in Ethereum blockchain. Ether can be transferred among accounts and exchanged to other currencies. 1 ether will be exchanged for each US$217 (recorded at Aug 18, 2019).

[3] Price per each unit of gas is determined by the sender. The higher the price, the faster the transaction may be processed. All consumed gas must be paid in ether.

In this contract, `Alice` is able to call to an arbitrary contract deployed at address `c` with an empty signature, which means invoking the fallback of recipient, and unlimited gas. For re-withdrawal prevention, `sent` is meant to be set to `true` after `c` successfully withdraws 1 ether. However, before switching to the terminated state, the fallback of `Fraudster` re-enters the `send` function of `Alice` and forces her to transfer ether endlessly, since `sent` still has value equal to `false`.

3.2 Gasless Send

A `send` function can be used in order to transfer ether among accounts with a gas stipend equals to 2300 units. However, this limited amount of gas is only enough to perform a single instruction (transferring ethers in this case) and cannot used for executing more complicated business logic that may lead to an "*out-of-gas*" exception. Different from `c.call.value(amount)()` at which the callee's signature is empty and all remaining gas is passed, `send` and `call` will return `false` to the caller contract (instead of throwing an exception and terminating transaction execution), and continue to execute the rest of instructions after the call to `c` with the remaining gas. Note that, the return values of `call` and `send` are often disregarded by developers; therefore, the remnant of code continues to execute at risk without any restriction.

3.3 Force-Sending Ether by Suicide

A smart contract is able to self-destruct for cleaning up all related information about itself in the blockchain data, including its bytecode and storage. These pieces of data will no longer exist from this point. And after the moment of self-destruction, this contract will transfer all ether that it is holding to another pre-defined contract.

Unlike a regular call, `selfdestruct` function does not invoke the fallback function of the recipient, but the balance of recipient is altered without any restriction. Assume that, the recipient is not allowed to receive any ether to prevent some adverse functions can occur. For example, a contract may discard all incoming ether transfers to it by reverting the transactions which trigger its fallback function. This is because some negative impacts may happen if the contract balance contains a positive value.

3.4 Integer Overflow

In Ethereum smart contract, balance of an arbitrary account is presented by an unsigned 256-bit integer denoted as `uint256`. After transferring, balances of the sender and receiver will be updated respectively. Because operations are being done in the 32-byte integer field, an integer overflow can happen in case there is no verification. Consider a contract (illustrated in Listing 3), where Alice will be able to withdraw some ether if her balance satisfies to some conditions. However,

if the conditions are set unrestrictedly, she still can proceed her `transfer` call. In Ethereum, let imagine a 1-byte integer works as an odometer, clocking from 0 to 255 and rollover to 0 afterwards. Hence, in 32-byte field, the operation of 0–1 yields a result of `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF` $\approx 3.4 \times 10^{38}$, instead of -1. This means that the business logic of smart contract works unintendedly since Alice's withdrawal draft is not rejected. Alice now, therefore, can endlessly withdraw 1 ether from the contract until contract balance runs out at 0.

```
function withdraw() {
    if (balance[alice] > 1) {
        balance[alice]--;
        alice.transfer(1);
    }
}
```

Listing 3. Contract contains integer overflow

3.5 Array Overflow

Recall that contracts in Ethereum hold the root of a Merkle Patricia tree in which all permanent data of the corresponding contract is stored. Each node in the tree is identified by a 256-bit index. Each contract, therefore, has ability to store up to 2^{256} 32-byte ***virtual* storage slot**s. It seems that collision cannot happen since variables are stored in distinct slots.

However, Hoyte [12] showed that there may exist unexpected data overridden if a dynamic array is declared in contract. In Listing 4, `isAttacked` and `map` variables are stored at `0x00..00` and `0x00..01` slot (64 hex-character in length), respectively. Since `map` is a dynamically allocated variable, its elements must be stored somewhere else which is nonconsecutive to two mentioned variables. The index of the first element of `map` is `map[0]` $\rightarrow$ `KECCAK256(index_of_map) = KECCAK256("00...01") = 0xd874...2827`, and the rest follows this slot. Because the dynamic array has no upper-bound for its number of elements, the index of element will increase linearly as the array expands. This results in an element will be allocated at index of `0xFF...FF` (64 F's). The next element will be in the `0x00..00` slot (slot indexes are also 256-bit integer) which are collided to value of `isAttacked`.

```
contract ArrayOverflowBug {
    bool public isAttacked;
    uint256[] map;

    function set(uint256 key, uint256 value) public {
        // Expand dynamic array as needed
        if (map.length <= key) {
            map.length = key + 1;
        }

        map[key] = value;
    }
}
```

Listing 4. Contract contains array overflow

3.6 Uninitialized Storage Pointer

In EVM, both memory and storage are used for handling stored value of contract variable. According to Ethereum's specification, a variable declared outside of a function are stored in storage by default. Meanwhile, the position where to store in-function variable is determined by the corresponding variable type. In details, elementary type variables (e.g. `int`, `bytes`, `bool`, etc.) are stored in memory; other complex types such as `array` and `struct`, however, have their value in storage.

```
1  contract StructStorageBug {
2      struct Donation {
3          uint256 timestamp;
4          uint256 etherAmount;
5      }
6      Donation[] public donations; // slot 0
7      address public owner;        // slot 1
8
9      function donate(uint256 etherAmount) public payable {
10         Donation donation;
11         donation.timestamp = now;
12         donation.etherAmount = etherAmount;
13
14         donations.push(donation);
15     }
16 }
```

Listing 5. Contract contains uninitialised storage pointer

Related to this innate of EVM, Beyer [4] explained a complex attack exploiting this dangerous behaviour. Consider a sample contract in Listing 5. If a donor transfers ether to this contract, it will record the amount and time the donation would have been sent. All donations are handled by an array of structures `donations` which is stored at slot `0` of storage. In addition, the fundholder has her address declared at slot `1`. This `owner` address is required to perform some severe tasks such as drawing on the fund and must not be amended unless all pre-defined conditions are satisfied. However, notice that a local variable `donation` is declared at Line `10`, whose type belongs to complex types. Hence, `donation` is not stored in memory but becomes a pointer to storage. Note that, the `donation` variable is not initialized resulting being allocated at slot `0` of storage by default, instead of at memory. Hence, in the next following line, the assignment to `donation.timestamp` will be essentially overriding over the `donations` variable. Similarly, `owner` will be overridden by the assignment to `etherAmount` property. In order to gain permission to this contract, an adversary could simply make a donation with his account address as parameter.

3.7 Overridden by Delegate Call

Since smart contracts have limitation in code size at approximately 24 KB [7], they may need to recycle functions from other contracts as using libraries in program development. Solidity allows contract to invoke functions of other contracts via `delegatecall`s whereas the context of storage, information of `msg.sender` and `msg.value` are preserved [6]. However, this call may be considered as a security risk for the caller contract because it must trust the callee contract which is permitted to commit changes on caller's storage [9].

In many cases, the address of caller is immutably determined in the callee contract source code. Problems will occur if this address can be manipulated by an adversary. Note that, all variables in the caller are treated as pointers to the callee's storage, which means all amendments during callers' execution will affect on callee's storage without any verification. The detailed explanation for a historical attack to "multi-sig" wallet Parity was described in [20].

4 Detection Methods for Attack Vectors

In this section, we introduce our detection methods for each type of attack vectors. In simple terms, smart contracts may contain vulnerabilities, which cause contracts to run on an unplanned scenario. However, these vulnerabilities are still harmless until an adversary takes advantage by exploiting them. Generally, he must send transactions, which are termed as attack vectors in security field, to exploit these vulnerabilities. These attack vectors are divided into categories, which will be explained in detail in Subsect. 4.1, according to the tracks that they left during the execution of attacking. The detailed explanation of the methods for detecting these attacks can be found in the subsections following the classification.

4.1 Category of Attack Vectors

In order to establish categories for assigning the known attack vectors, we show typical examples to illustrate how vital evidence left whilst attacking. In terms of **transaction result**s, the **final result**, which is comprised of the shift in involved addresses balance, consumed gas amount, etc., can be used to label the category. For example, the average gas amount used to proceed a transaction in Ethereum blockchain is approximately 21,000 units. This common number of gas usage can help detect any transaction consuming a huge amount of gas. Besides, the **intermediate result** calculated during transaction execution, including temporary values of all operations in stack and memory of smart contract, can lead to a successful attack. Consider a step in stack using `SUB` operator with two corresponding parameters `0` and `1`, this operation leads to a result of `0xFFFF..FF` (64 F's) since integer overflow happens here. This unusual output can be worse if these numbers, respectively, present for an adversary tried to send `1` unit of token, while her balance was at `0` but the receiver got an unacceptably high amount of value.

In another aspect, each contract is able to store 2^{256} 32-bytes-long slots, which form contract **storage**. Any modifications in the storage also help to recognize whether the smart contract has been exploited. Some attacks seemed to alter the storage values to unusual ones. The irregularity is presented in **value of variable**, where i.e., the address of an immutable contract `owner` has been changed to an extraordinary address. For another example, the balance of an account is often declared in `balance` variable. If this variable contains an extremely high value, we may consider it as being modified by an unusual

behaviour. Moreover, changes in storage can be counted as abnormal regarding to **type of variable**. In some attack scenarios, the storage slots are overwritten by values whose type mismatches to the declared type of corresponding variables. For instance, assume that `uint256` variable holds the value of `0x546869734973415374726696e6700..00` which infers to a meaningful string—`ThisIsAString`. In such cases, we can imply that there have occurred abnormal transactions.

Table 1. Category of attack vectors according to their tracks

Level	Attack vector	Result		Storage	
		Final	Internal	Value	Type
Solidity	Reentrancy	✓	✓	—	—
	Gasless send	—	✓	—	—
	Force-sending ether by suicide	—	✓	—	—
EVM	Integer overflow	—	✓	—	—
	Array overflow	—	✓	✓	✓
	Uninitialised storage pointer	—	✓	✓	✓
	Overridden by delegate call	—	✓	✓	✓

To sum up, our category of attack vectors consists of the unusual value in final and intermediate results of the transaction; and the extraordinary modification in type and value in the storage of the involved smart contract. Table 1 represents our classification. Recall that the findings of these attack vectors along with their description are collected from [2] and many security blogs and will be represented in the next following subsections.

Additionally, based on possible methods to detect whether an incoming transaction is an attack vector, detecting each attack vector requires vary inputs. During experiments, we found that there are three valuable factors may be used as input for the detection. For details, we need to perform the detection algorithms on only the newest transaction or all previous transactions. Moreover, only contract bytecode is involved in the detection stage, or we also need the information of corresponding source code. And lastly, the storage is whether used as input data to detect the attack. Table 2 summarizes the necessary inputs for detecting each type of attack vectors.

4.2 Reentrancy

During the exploitation of this attack, a chain of nested internal function calls was created, in which internal calls have a same information of sender and recipient. In order to capture the attack, we re-execute the transaction and verify if there exists any nested internal calls with duplicated patterns. Note that, the duplicated calls do not need to be called right after another call, but can be squeezed in by some manipulated calls created by the attacker.

Table 2. Inputs for detecting each type of attack vectors

Attack vector	Transaction	Contract	Storage
Integer overflow	Current	Bytecode	No
Reentrancy	Current	Bytecode	No
Gasless send	Current	Bytecode	No
Force-send ether by suicide	Current	Bytecode and sourcecode	No
Array overflow	Current	Bytecode	No
Uninitialised storage pointer	All	Bytecode and sourcecode	Yes
Overridden by delegate call	All	Bytecode and sourcecode	Yes

4.3 Gasless Send

The detection method for this kind of attack is similar to *reentrancy* case. Note that, in *gasless send* attack vector, there exists at least a failed internal transaction while its parent transaction executes successfully. At EVM bytecode level, an internal transaction is considered as success if its last instruction is either `STOP` or `SSTORE` [16]. Meanwhile, an internal transaction fails to execute when reach to these following conditions (see Fig. 1):

- Function call is reverted by `assert()` function and generates `INVALID` instruction;
- Function call is reverted by either `revert()` or `throw()` and generates `REVERT` instruction;
- Execution is terminated at an arbitrary step where gas stipend is all consumed, resulting in a field named `error = {}` generated in debug logs.

To detect this kind of attack vector, we track through nested calls, starting from the deepest transaction in debug logs, and apply these following steps.

Step 1. Verify whether current transaction is reverted, according to above-listed conditions;
Step 2. Verify whether ancestor transaction is consequently reverted:

- If ancestor transaction is an internal call, we will apply Step 1 to this transaction;
- If ancestor transaction is a normal transaction, we will need to make sure that this transaction executes successfully, described by `false` value in `failed` field of debug calls.

4.4 Force-Sending Ether by Suicide

In this kind of attack vector, we consider the recipient contract as the supervised target (instead of the contract committing suicide). Note that, `payable` keyword is used to declare that this contract ignores all ether. Therefore, we detect suicide to force-send ether by two steps:

(a) Successfully executed internal transaction

(b) Unsuccessfully executed internal transaction

Fig. 1. Stack states of internal transaction

- Generating abstract syntax tree (AST) while compiling sourcecode. Checking value of `payable` field of fallback function.
- Re-executing all transactions in blockchain, and filtering `SELFDESTRUCT` instruction with its corresponding recipient address. Verifying if this address matches address of supervised contract.

4.5 Integer Overflow

The root cause of integer overflow vulnerabilities is because all arithmetic operations are not checked for overflow at the EVM level. However, this instinct is not improperly developed due to two important purposes. Firstly, reducing the number of computational steps that allows users to pay a smaller fee of gas. Secondly, EVM needs overflowed results for some specific tasks. For example, to invert `0x00001010` to `0x11110101`, EVM will start at `0xFFFFFFFF = 0 - 1` and calculate `0x00001010 XOR 0xFFFFFFFF` to get the desired result. In an Ethereum improvement proposal, Alex Beregszaszi [3] listed all possible cases lead to integer overflow, including:

- `DIV` and `SDIV` with a zero divisor;
- `ADD`, `MUL`, `EXP` equals to a result whose length is greater than 256 bits;
- `SUB` when the subtrahend exceeds minuend;
- `SDIV` when -2^{255} divided by -1;
- `ADDMOD` and `MULMOD` with mod equals 0;
- `SIGNEXTEND` when the parameter of position is greater than 31.

In terms of detecting this integer overflow attack, we catch all unexpected overflowed results appearing during transaction execution satisfying the above-mentioned conditions. However, all exploitation related to this attack are found to be invoked by a function along with extraordinary parameter values, which are all controlled by the adversary. In order to improve the detection algorithm, we introduce tracer aiming at backtracking over each step in the stack if an overflowed value has origin from user's input.

Figure 2 illustrates how stack state changes after each instruction. In the upper image, the result from `ADD` op-code is considered as overflow due to two reasons, (i) the first parameter `00000022` is calculated from a value loaded by `CALLDATALOAD`, and (ii) the result is truncated to a 4 bytes value. Meanwhile, in the lower image, although `ADD` gives a result which is overflowed, two input parameters do not originate in a `CALLDATALOAD` opcode but are pushed directly from the contract's bytecode via `PUSHx` instead.

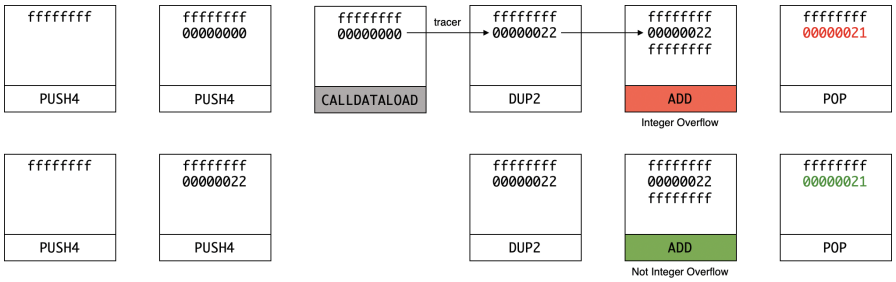

Fig. 2. Example of stack trace in integer overflow

4.6 Array Overflow

The detection method for this case recycles results from *integer overflow*. Indeed, starting from instructions detected as related to integer overflow attack vector, we track straightforwardly through stack states to capture `SSTORE` instructions which use overflowed value to determine the index of slot to modify (see Fig. 3).

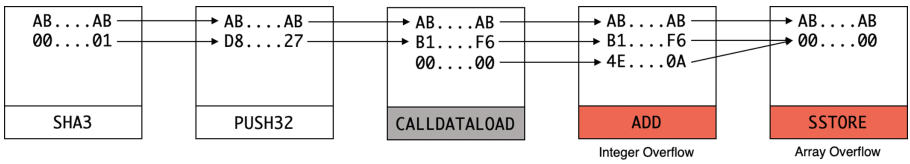

Fig. 3. Stack states of array overflow attack vector

4.7 Uninitialized Storage Pointer

It is important to note that all data transfers (reading and writing) among stack, memory and storage are done over 32-bit field. However, there is no type checking is done for all operations in EVM, including those transfer operations. Because of that, type-mismatching between variable type and variable usage context can occur. For instance, an arbitrary `address` variable as `owner` in the previous example can easily overridden by a `uint256` value. In fact, according to Solidity documentation[4], each type of variable in EVM has different layout of state variable. Therefore, to detect whether a write-access is trying to override the storage, we propose a heuristic based on involved constructs' type.

On purpose of implementing this heuristic, we need the help of a storage recreator for each contract. From the input of AST and amendments on each transaction, we can recover the information of each byte in storage including its corresponding type, value and to which variable this byte belongs. On each transaction altering contract storage, we will mark it as a suspicious transaction if one of the following condition is violated:

1. Unallocated bytes i.e. bytes that present the value of no variable, are written illegally;
2. A slot belonging to a `mapping` variables is altered;
3. A slot of a non-string variable contains an in-string-format value;
4. A huge value is stored in a slot that represents value of a dynamic array.

Finally, the algorithm to detect this kind of attack is described as follow.

Step 1. Filtering all transactions interacting to the supervised contract;
Step 2. On each transaction, seeking for `SSTORE`:

- If this operation is performed on an untracked slot, marking its corresponding type;
- Else, verifying this operation according to above-defined conditions.

4.8 Overridden by Delegate Call

The detection method for this attack vector is similar to *Uninitialized storage pointer* (see Subsect. 4.7) since it is based on storage overrides. The only difference is we need to verify whether the `SSTORE` is performed in the context of a `DELEGATECALL` whose caller is the supervised contract.

5 System Architecture

In order to investigate attack vectors sent to smart contracts in Ethereum blockchain, we introduce Abbe—the abnormal behaviours detecting tool. Our tool takes several information of contracts as input parameters as regards type of

[4] https://solidity.readthedocs.io/en/latest/miscellaneous.html#layout-of-state-variables-in-storage.

attack vectors. This section presents our proposed architecture for ABBE, which is comprised of five components as illustrated in Fig. 4.

Contract Initialization. At the first step, the Contract Initialiser module gets *address* and *sourcecode* with respect to the supervised contract for initialization purpose. Users must specify the *address* at the beginning. Meanwhile, this module will seek for the *source-code* of contract in Etherscan. Etherscan is the de facto location for exploring and seeking Ethereum transactions, tokens, smart contracts, etc. The source-code that belongs to an arbitrary smart contract is committed by the developers and is double-checked on Etherscan by comparing two versions of bytecode, one is fetched from Ethereum blockchain and the other one is taken after compiling the submitting sourcecode. In case of the source-code is missing in Etherscan, another option is preferred, a local path leading to *sourcecode* must be specified by the users.

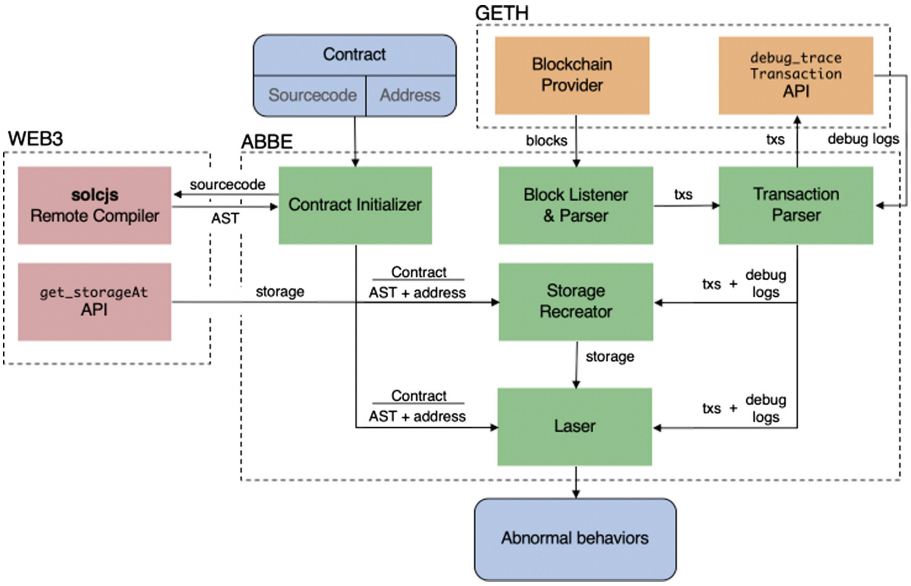

Fig. 4. System architecture of ABBE

Block Parsing and Transaction Parsing. After loading, this module continues to compile sourcecode of the smart contract into bytecode form. During this step, the compiler generates an abstract syntax tree containing information about type, scope, and other properties of all tokens in contract source. Note that there are nearly 50 versions of Solidity compiler which may be used to compile the contract source code. Because of this massive amount of releases, instead of importing all releases of compiler, the Initialiser finishes this step with

the help of a remote compiler `solcjs`[5], featured by `web3` library, for the sake of convenience.

Simultaneously, the Block Listener and Parser module serves as a listener that fetches new blocks from the Blockchain Provider. Blockchain Provider essentially is the main application programming interface (API) of an Ethereum client, which is GETH in this tool. All new blocks are generated, processed by GETH and are emitted to all connections that GETH holds via its main API. On input the new block, this module parses it into block header along with all transactions grouped inside. Each parsed transaction will be processed by the Transaction Parser module in the next step.

Transaction Parser. This module takes all information of each transaction as its input, and then filters out the data of hash, sender address, receiver address, consumed gas amount, etc. The use of this information helps call the `debug_traceTransaction` API to generate corresponding debug logs. This API is provided by the GETH provider mentioned in the previous step.

Storage Recreation. Following these steps, the relevant information of smart contracts including the *AST* of contract and *debug logs* of all executed transactions (regardless of whether their transaction type is normal or internal) from genesis block is used as input for the Storage Recreator module. At this time, the Storage Recreator creates a mapping table of variables' name, type, and slot storage index where its value stored.

Transaction Diagnosis. Finally, the tool passes data of *AST*, *address* and *storage* of the contract, along with all *transactions* attached with their *debug logs* into **Laser**—the diagnosing module. On each incoming transaction, this module matches it to the pre-defined attack vectors patterns, and returns whether this transaction is considered as an abnormal behaviour.

6 Implementation and Testing

According to the architecture introduced in Sect. 5, we successfully implemented the Abbe tool[6,7] and performed testing in our private Ethereum network. Since our implementation requires an archive node of GETH client, the large-scale experiment of Abbe in Ethereum mainnet will be carried out in future work.

To start, users have to determine the address of contract and specify the local path leading to contract sourcecode, if exist. Users are also allowed to configure the number of block where Abbe starts to perform detection. A snapshot of a sample result given by Abbe is illustrated in Fig. 5. On each input transaction,

[5] All releases of `solcjs` are listed at https://ethereum.github.io/solc-bin/bin/list.js.

[6] The private repository of Abbe is located at https://github.com/nxqbaos/abbe2. Access to this repository is granted upon requests.

[7] The Abbe tool has been invited to be presented at Hack In The Box (HITB+) Cyber Week's Conference, *Oct. 15–17, 2019*, Abu Dhabi, UAE.

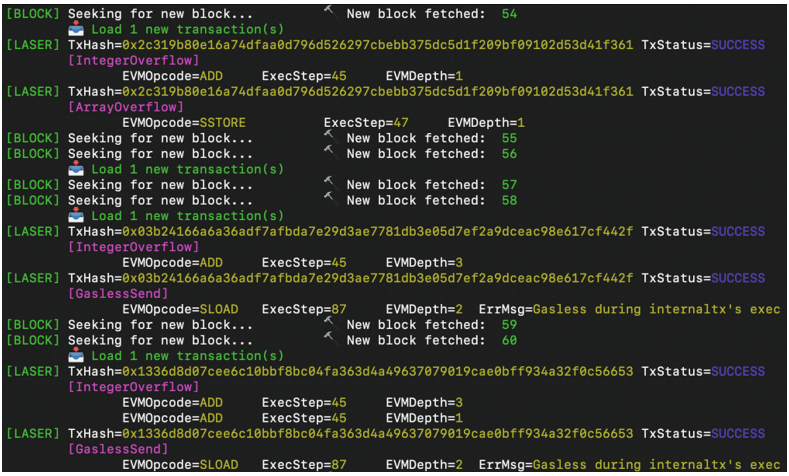

Fig. 5. Sample result given by ABBE

if there is any abnormal behavior detected, ABBE will put out an alert. The format of output is as follows. First, the `LASER` yields the hash of the transaction that performs attack and its final state, which is `FAILURE` or `SUCCESS`. The next following lines represent the name of the detected attack vectors and their corresponding information. This information consists of different fields according to each type of attack vector. Generally, these fields include:

- `EVMOpcode`: Instruction mnemonic.
- `ExecStep`: The execution step counter in the debug log.
- `EVMDepth`: The depth number of the current call in the list of nested call. This number of depth starts from `1`, equivalent to the origin call which is triggered from a normal transaction.
- `ErrMsg`: The cause that leads to failure of internal transaction (in `GaslessSend` case); or, the cause leading to override in contract storages (in `DelegateCall` and `UninitPointer` case).

The test suite that we used for testing ABBE comprises of contracts containing vulnerabilities, or have been attacked in the past, or have been collected from [2,22,28]. On each attack vector, we performed multiple test and visualize the result in the confusion matrix form.

Especially, as for three attack vectors *reentrancy*, *gasless send* and *forced-sending ether by suicide* in the Solidity level, the tool is able to fully detect all transactions exploiting these three vulnerabilities. The confusion matrix in Table 3 shows that there is no false detection occurring during the tests.

However, while performing test on smart contracts that contain *reentrancy* vulnerability, the tool detects that there also exist the *gasless send* attack. This is because the pattern of *gasless send* is similar to the *reentrancy* case. Recall that, a transaction related to the *reentrancy* behaviour recursively calls a specific function until gas is all consumed. When this transaction creates the call

Table 3. Testing results in Solidity level cases

		Reentrancy		Gasless send		Suicide to force-send	
		Pos	Neg	Pos	Neg	Pos	Neg
Result of Abbe	Pos	**10**	0	**29**	0	**10**	0
	Neg	0	**7**	0	**10**	0	**0**

which is the most-inner internal transaction, the gas that passed into this call is insufficient to execute that function properly. Hence, we consider all *gasless send* alerts caused by *reentrancy* is valid.

Besides, for early awareness of being attacked, we also record all transactions which attempt to perform *gasless send* attack but result in failure. To achieve this, Abbe marks the transactions having the attack pattern of this case, in other words, there exists an internal transaction which got an insufficient gas stipend, although the transaction ended up in a `failed` state.

Table 4. Testing results in overflow cases

		Integer overflow		Array overflow	
		Pos	Neg	Pos	Neg
Result of Abbe	Pos	**30**	3	**13**	1
	Neg	28	**36**	0	**12**

Meanwhile, as for *integer overflow*, the result in the Table 4 shows that our Abbe tool produces positive result as it passes most of test cases. However, there are 28 test cases that lead to false negative, since the tool only defines integer overflow pattern on the field of 32-bit integer. The detection of overflow in fields whose bit-size is smaller will be supported in further work. A small number of false positive is recorded. These false positive results happen when there exists reentrancy in the transaction.

As for the *array overflow* case, the result achieved is apparently impressive since there is only 01 false positive left. In this test case, the input does create an intermediate result which is arithmetic overflowed. Recall that our algorithm detects array overflow by verifying whether the calculations of slot index that to be written are overflowed. The overflow of intermediate result, however, is equal to the index of slot that need to be written in the storage, while the calculations for identifying the slot indexes are not related to the input values.

Lastly, the results obtained when applying the heuristic to detect two attack vectors related to overriding on EVM storage is presented in Table 5. Although Abbe can recognise attacks in many testcases, our tool still fails under some tricky inputs.

Table 5. Testing results in heuristic-based cases

		Delegated call		Uninitialised pointer	
		Pos	Neg	Pos	Neg
Result of ABBE	Pos	**8**	1	**8**	1
	Neg	13	**9**	11	**10**

The false negative results are all caused by the mishandling the verification in types of value. In our proposed algorithm, the detecting pattern checks whether the actual type of the input value and the expected type of the accessing slot are mismatched. Due to this reason, if the input value and the slot have the same type, our tool cannot detect the abnormality occurred. Besides, if the slot containing data of some variables which are padded together into a form of 32 bytes, e.g. `uint16=0x6464`, `uint232=0x0`, `bool=true` sequentially, is overridden by a string, ABBE also cannot catch this illegal write access since the string has the value of `646400...0001` which matches the value-format of that slot. Generally, ABBE has a drawback in detecting attacks that exploit mismatches in types. However, this major drawback is because the checks for arithmetic instructions in EVM is not accomplished.

The only false positive is produced when an arbitrary variable is set to an extremely large value. Especially, users can set `array_name.length` to a huge value for some specific purposes. However, this situation may not happen in real contracts since `array_name.length` regularly cannot be changed manually, or altered to a huge value as in our test.

7 Related Work

Taxonomy of Vulnerabilities. Our work is guided by the previous taxonomy of Atzei *et al.* [2], the first classification that divides Ethereum smart contract attacks into three categories according to three levels of smart contracts. Since this work published, many new vulnerabilities have been reported and described by many personal blogs. Therefore, we append those new vulnerabilities that our tool can detect into the existed taxonomy.

Smart Contract Security Analysis. To the best of our knowledge, there has not existed any work investigating attacks detection in smart contracts so far. The major attacks recorded in the past few years were all discovered manually.

In efforts to feature smart contract security analyses, many security teams provide solutions to tackle this problem. The project introduced by SmartCheck [21] supports detecting vulnerabilities in the smart contracts' source code based on the defined bug patterns. Because this tool performs analysis on only Solidity level, the vulnerabilities that can be detected are independent of contracts' logic flaws. OYENTE, introduced by Luu *et al.* [14], is a security analysis tool that relies on both contracts' source code and bytecode to perform symbolic execution. Moreover, OYENTE aims to detect four specified vulnerabilities, including

integer overflow, transaction dependency, stack-depth and reentrancy. On the other hand, our tool focuses on detecting the exploit transactions and in a wider range of vulnerabilities.

In the same vein with Oyente, the tool Maian [15], Mythril [17], Manticore [19] and Securify [26] perform analyses for vulnerabilities in smart contracts by using symbolic execution on EVM bytecode. This approach, however, produces all possible condition paths as all logic flaws that the contract execution can reach. As the size of contracts enlarges, these tools need to verify a larger number of paths, the consumed-time for security analysis therefore increase. On the other hand, Abbe not only reason about EVM bytecode but it also analyses Solidity source code of the contracts since each vulnerability is expressed differently in these two levels.

Tann *et al.* [25] used a divergent method that applied sequence learning to detect security threats in smart contracts. By using long-short term memory neural network, this work introduced a machine learning technique that processes over smart contract bytecode. After generating the training set from over 640,000 distinct contracts which are labelled as safe or vulnerable by the Maian tool [15], the authors claimed their proposed model obtain a surprising result and provide improvements over symbolic analysis methods. Although Abbe currently follows the traditional approach, the idea of analysis by machine learning for detecting attacks is a viable research direction to follow.

Formal Verification. In addition to vulnerability detection, there have also been works on checking smart contracts against user-defined policy. F* [5] chooses a subset of EVM and Solidity to translate bytecode and source code of the smart contracts into F* respectively. This translated program is then verified by the F*-based verifier against the defined-assertions of users. However, the capability of F* is limited at processing over a small subset of EVM which does not contain loop. In another work of Kalra *et al.* [13], the formal verification is supported by Zeus framework. Similar to F*, Zeus also translates both smart contracts' source code and bytecode into an intermediate representation based on LLVM, then starts the verification by using an SMT-based solver.

8 Conclusion

This paper expands the taxonomy of vulnerabilities in smart contracts with new types of attacks. For each vulnerability, we also propose methods to detect its attack vectors and demonstrated them in Abbe—the detecting tool. Abbe can detect seven kinds of attacks with good performance. Especially, our tool can detect all attacks which related to vulnerabilities in Solidity level and some other kinds of attack with zero false positives. Because our methodology is based on verifying all modifications on smart contract storage, not only the known attacks can be detected, but zero-day attack patterns can also be discovered.

For the intermediate future, we are working on conducting tests at a larger scale on the mainnet of Ethereum and other blockchain networks that support

smart contracts written in Solidity. In order to potentially identify undefined-attacks, we also plan to apply data mining techniques based on the previous transactions in the contract's history. Further work using data classification model may allow security threats that exploit smart contract vulnerabilities can be detected in a higher efficiency.

Acknowledgement. During the preparation of this work, the first author was partially supported by University of Technology (HCMUT), VNU-HCM under "Student Scientific Research" Grant Number 121/HĐ-ĐHBK-KHCN&DA; and the last author was partially funded by Vietnam National University-HCMC under Grant C2019-20-14. The authors would like to thank Nguyen Van Thanh for his comments helping to improve the manuscript significantly.

References

1. Post-Mortem Investigation (2016). https://www.kingoftheether.com/postmortem.html
2. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
3. Beregszaszi, A.: EVM: overflow detection in arithmetic instructions (2016). github.com/ethereum/EIPs/issues/159
4. Beyer, S.: Storage allocation exploits in ethereum smart contracts (2018). https://medium.com/cryptronics/storage-allocation-exploits-in-ethereum-smart-contracts-16c2aa312743
5. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, pp. 91–96. ACM (2016)
6. Buterin, V.: Ethereum Improvement Proposal 7 (2015). https://github.com/ethereum/EIPs/blob/master/EIPS/eip-7.md
7. Buterin, V.: Ethereum Improvement Proposal 170 (2016). https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md
8. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. White Paper (2014)
9. Buterin, V., et al.: Difference between CALL, CALLCODE and DELEGATECALL (2016). https://ethereum.stackexchange.com/questions/3667/difference-between-call-callcode-and-delegatecall
10. Consensys: Solidity Recommendations (2018). https://consensys.github.io/smart-contract-best-practices/recommendations/
11. Falkon, S.: The story of the DAO - its history and consequences (2017). https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee
12. Hoyte, D.: MerdeToken: it's some hot shit (2018). https://github.com/Arachnid/uscc/tree/master/submissions-2017/doughoyte
13. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: analyzing safety of smart contracts. In: NDSS (2018)
14. Luu, L., et al.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269. ACM (2016)

15. Manticore (2018). https://github.com/trailofbits/manticore
16. McKie, S.: Solidity learning: Revert(), Assert(), and Require() in solidity, and the new REVERT Opcode in the EVM (2017). https://medium.com/blockchannel/the-use-of-revert-assert-and-require-in-solidity-and-the-new-revert-opcode-in-the-evm-1a3a7990e06e
17. Mueller, B.: Mythril - Reversing and Bug Hunting Framework for the Ethereum Blockchain
18. Nakamoto, S., et al.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
19. Nikolić, I., et al.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 653–663. ACM (2018)
20. Palladino, S.: The parity wallet hack explained - zeppelin blog (2017). https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7
21. SmartDec: automatically checking smart contracts for vulnerabilities and bad practices (2018). https://tool.smartdec.net
22. SMARX: Capture the ether - the game of ethereum smart contract security (2018). https://capturetheether.com
23. SpankChain: We Got Spanked: What We Know So Far (2018). https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe
24. Szabo, N.: Smart Contracts. Unpublished manuscript (1994)
25. Tann, A., Han, X.J., Gupta, S.S., Ong, Y.S.: Towards safer smart contracts: a sequence learning approach to detecting vulnerabilities (2018). arXiv preprint arXiv:1811.06632
26. Tsankov, P., et al.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82. ACM (2018)
27. Wood, G., et al.: Ethereum: A Secure Decentralised Generalised Transaction Ledger. Ethereum project yellow paper **151**, 1–32 (2014)
28. Zeppelin team: The Ethernaut Wargame. https://ethernaut.zeppelin.solutions