sCompile: Critical Path Identification and Analysis for Smart Contracts

Jialiang Chang*, Bo Gao†, Hao Xiao†, Jun Sun† and Zijiang Yang*
*Department of Computer Science
Western Michigan University, Kalamazoo, MI, USA
Email: {jialiang.chang||zijiang.yang}@wmich.edu
†Pillar of Information System Technology and Design
Singapore University of Technology and Design, Singapore
Email: bo_gao@mymail.sutd.edu.sg, {hao_xiao||sunjun}@sutd.edu.sg

arXiv:1808.00624v1 [cs.CR] 2 Aug 2018

***Abstract*—Smart contracts are an innovation built on top of the blockchain technology. It provides a platform for automatically executing contracts in an anonymous, distributed, and trusted way, which has the potential to revolutionize many industries. The most popular programming language for creating smart contracts is called Solidity, which is supported by Ethereum. Like ordinary programs, Solidity programs may contain vulnerabilities, which potentially lead to attacks. The problem is magnified by the fact that smart contracts, unlike ordinary programs, cannot be patched easily once deployed. It is thus important that smart contracts are checked against potential vulnerabilities.**

Existing approaches tackle the problem by developing methods which aim to automatically analyze or verify smart contracts. Such approaches often results in false alarms or poor scalability, fundamentally because Solidity is Turing-complete. In this work, we propose an alternative approach to automatically identify critical program paths (with multiple function calls including *inter-contract* function calls) in a smart contract, rank the paths according to their criticalness, discard them if they are infeasible or otherwise present them with user friendly warnings for user inspection. We identify paths which involve monetary transaction as critical paths, and prioritize those which potentially violate important properties. For scalability, symbolic execution techniques are only applied to top ranked critical paths. Our approach has been implemented in a tool called sCompile, which has been applied to 36,099 smart contracts. The experiment results show that sCompile is efficient, i.e., 5 seconds on average for one smart contract. Furthermore, we show that many known vulnerability can be captured if the user inspects as few as 10 program paths generated by sCompile. Lastly, sCompile discovered 224 unknown vulnerabilities with a false positive rate of 15.4% before user inspection.

I. Introduction

Built on top of cryptographic algorithms [1], [2], [3] and the blockchain technology [4], [5], [6], cryptocurrency like Bitcoin has been developing rapidly in recent years. Many believe that it has the potential to revolutionize the banking industry by allowing monetary transactions in an anonymous, distributed, and trusted way. Smart contracts bring it one step further by providing a framework which allows any contract (not only monetary transactions) to be executed in an autonomous, distributed, and trusted way. Smart contracts thus may revolutionize many industries. Ethereum [7], an open-source, blockchain-based cryptocurrency, is the first to integrate the functionality of smart contracts. Due to its enormous potential, its market cap reached at \$45.13 billion as of Nov 28th, 2017 [8].

In essence, smart contracts are computer programs which are automatically executed on a distributed blockchain infrastructure. Popular applications of smart contracts include crowd fund raising and online gambling, which often involve monetary transactions as part of the contract. Majority of smart contracts in Ethereum are written in a programming language called Solidity. Like ordinary programs, Solidity programs may contain vulnerabilities, which potentially lead to attacks. The problem is magnified by the fact that smart contracts, unlike ordinary programs, cannot be patched easily once they are deployed on the blockchain.

In recent years, there have been an increasing number of news reports on attacks targeting smart contracts. These attacks exploit security vulnerabilities in Ethereum smart contracts and often result in monetary loss. One notorious example is the DAO attack, i.e., an attacker stole more than 3.5 million Ether (equivalent to about \$45 million USD at the time) from the DAO contract on June 17, 2017. This attack is carried out through a bug in the DAO contract. The correctness and systematic analysis and verification of smart contracts since then have been brought into sight with urgency.

There have been multiple attempts on building tools which aim to analyze smart contracts fully automatically. For instance, Oyente [9] applies symbolic execution techniques to find potential security vulnerabilities in Solidity smart contracts. Oyente has reportedly been applied to 19,366 Ethereum contracts and 45.6% of them are flagged as vulnerable. Another example is Zeus [10], which applies abstract interpretation to analyze smart contracts and claims that 94.6% of the contracts are vulnerable. In addition, there are approaches on applying theorem proving techniques to verify smart contracts which requires considerable manual effort [11].

The problem of analyzing and verifying smart contracts is far from being solved. Some believe that it will never be, just as the verification problem of traditional programs. Solidity is designed to be Turing-complete which intuitively means that it is very expressive and flexible. The price to pay is that almost all interesting problems associated with checking whether a smart contract is vulnerable are undecidable [12].

Consequently, tools which aim to analyze smart contracts *automatically* either are not scalable or produce many false alarms. For instance, Oyente [9] is designed to check whether a program path leads to a vulnerability or not using a constraint solver to check whether the path is feasible or not. Due to the limitation of constraint solving techniques, if Oyente is unable to determine whether the path is feasible or not, the choice is either to ignore the path (which may result in a false negative, i.e., a vulnerability is missed) or to report an alarm (which may result in a false alarm).

In this work, we develop an alternative approach for analyzing smart contracts. On one hand, we believe that manual inspection is unavoidable given the expressiveness of Solidity. On the other hand, given that smart contracts often enclose many behaviors (which manifest through different program paths), manually inspecting every program path is simply overwhelming. Thus, our goal is to reduce the manual effort by identifying a small number of critical program paths and presenting them to the user with easy-to-digest information. Towards this goal, we make the following contributions in this work.

1) We develop a tool called sCompile. Given a smart contract, sCompile constructs a control flow graph (CFG) which captures all possible control flow including those due to the *inter-contract* function calls. Based on the CFG, we can systematically generate program paths which are constituted by a bounded sequence of function calls.
2) As the number of program paths are often huge, sCompile then statically identifies paths which are 'critical'. In this work, we define paths which involve monetary transaction as critical paths. Focusing on such paths allows us to "follow the money", which is often sufficient in capturing vulnerabilities in smart contracts.
3) Afterwards, to prioritize the program paths, sCompile analyze each path to see whether it potentially violates certain critical property. We define a set of (configurable) money-related properties based on existing vulnerabilities. After the analysis, sCompile ranks the paths by computing a criticalness score for each path. The criticalness score is calculated using a formula which takes into account what properties the path potentially violates and its length (so that a shorter path is more likely to be presented for user inspection).
4) Next, for each program path which has a criticalness score larger than a threshold, sCompile automatically checks whether it is feasible using symbolic execution techniques. The idea is to automatically filter those infeasible ones (if possible) to reduce user effort.
5) Lastly, the remaining critical paths are presented to the user for inspection through an interactive user interface.

sCompile is implemented in C++ and has been applied systematically to 36,099 smart contracts which are gathered from EtherScan [13]. Our experiment results show that sCompile can efficiently analyze smart contracts, i.e., it spends 5 seconds on average to analyze a smart contract (with a bound on the number of function calls 3 including calls through inter-contract function calls). This is mainly because sCompile is designed to rank the program paths based on static analysis and only applies symbolic execution to critical paths, which significantly reduces the number of times symbolic execution is applied. Furthermore, we show that sCompile effectively prioritizes programs paths which reveal vulnerabilities in the smart contracts, i.e., it is often sufficient to capture the vulnerability by inspecting the reported 10 or fewer critical program paths. Lastly, using sCompile, we identify 224 vulnerabilities. The false positive rate of the identified property-violating paths (before they are presented to the user for inspection) is kept to an acceptable 15.4%. We further conduct a user study which shows that with sCompile's help, users are more likely to identify vulnerabilities in smart contracts.

The rest of the paper is organized as follows. Section II illustrates how sCompile works through a few simple examples. Section III presents the details of our approach step-by-step. Section IV shows evaluation results on sCompile. Section V reviews related work and lastly Section VI concludes with a discussion on future work.

II. Illustrative Examples

In this section, we present multiple examples to illustrate vulnerabilities in smart contracts and how sCompile helps to reveal them. The contracts are shown in Fig. 1.

Example 1: Contract *EnjinBuyer* is a token managing contract. It has 2 inherent addresses for *developer* and *sale*. In function *purchase_tokens()*, the balance is sent to the sale's address. There is a mistake on the sale's address and as a result the balance is sent to a non-existing address and is lost forever. Note that any hexadecimal string of length not greater than 40 is considered a valid (well-formed) address in Ethereum and thus there is no error when function *purchase_tokens()* is executed.

Given this contract, the most critical program path reported by sCompile is one which invokes function *purchase_tokens()*. The program path is labeled with a message stating that the address does not exist on Ethereum mainnet. With this information, the user captures the vulnerability.

Example 2: Contract *toyDAO* is a simple contract which has the same problem of the DAO contract. Mapping *credit* is a map which records a user's credit amount. Function *donate()* allows a user to top up its credit with 100 wei (which is a unit of Ether). Function *withdraw()* by design sends 20 wei to the message sender (at line 1) and then updates *credit*. However, when line 1 is executed, the message sender could call function *withdraw()* through its fallback function, before line 2 is executed. Line 1 is then executed again and another 20 wei is sent to the message sender. Eventually, all Ether in the wallet of this contract is sent to the message sender.

In sCompile, inspired by common practice in banking industry, users are allowed to set a limit on the amount

```
contract EnjinBuyer {
    address public developer =
        0x0639C169D9265Ca4B4DEce693764CdA8ea5F3882;
    address public sale =
        0xc4740f71323129669424d1Ae06c42AEE99da30e;
    function purchase_tokens() {
        require(msg.sender == developer);
        contract_eth_value = this.balance;
        require(sale.call.value(contract_eth_value)());
        require(this.balance==0);
    }
}

contract toyDAO{
    address owner;
    mapping (address => uint) credit;
    function toyDAO() payable public {
        owner = msg.sender;
    }
    function donate() payable public{
        credit[msg.sender] = 100;
    }
    function withdraw() public {
0       uint256 value = 20;
1       if (msg.sender.call.value(value)()) {
2           credit[msg.sender] = credit[msg.sender] - value;
        }
    }
}

contract Bitway is ERC20 {
    function () public payable {
        createTokens();
    }
    function createTokens() public payable {
        require(msg.value > 300);
        ...
    }
    ...
}
```

Fig. 1: Illustrative contracts

transferred out of the wallet of the contract. Assume that the user sets the limit to be 30. Given the contract, a critical program path reported by sCompile is one which executes line 0, 1, 0, and 1. The program path is associated with a warning message stating that the accumulated amount transferred along the path is more than the limit. With this information, the user is able to capture the vulnerability. We remark that existing approaches often check such vulnerability through a property called reentrancy, which often results in false alarms [9], [10].

Example 3: Contract *Bitway* is another token management contract. It receives Ether (i.e., cryptocurrency in Ethereum) through function *createTokens()*. Note that this is possible because function *createTokens()* is declared as *payable*. However, there is no function in the contract which can send Ether out. Given this contract, sCompile identifies a list of critical program paths for user inspection. The most critical one is a program path where function *createTokens()* is invoked. Furthermore, it is labeled with a warning message stating that the smart contract appears to be a "black hole" contract as there is no program path for sending Ether out, whereas this program path allows one to transfer Ether into the wallet of the contract. By inspecting this program path and the warning message, the user can capture the vulnerability. In comparison, existing tools like Oyente [9] and MAIAN [14] report no vulnerability given the contract. We remark that even although MAIAN is designed to check similar vulnerability, it checks whether a contract can receive Ether through testing[1] and thus results in a false negative in this case.

III. Approach

In this section, we present the details of our approach step-by-step. Fig. 2 shows the overall work flow of sCompile. There are six main steps. Firstly, given a smart contract, sCompile construct a control flow graph (CFG) [15], based on which we can systematically enumerate all program paths. Secondly, we identify the monetary paths based on the CFG up to a user-defined bound on the number of function calls. Thirdly, we analyze each program path in order to check whether it potentially violates any of the pre-defined monetary properties. Next, we compute a criticalness score for each program and rank the paths accordingly. Afterwards, we apply symbolic execution to filter infeasible critical program paths. Lastly, we present the results along with the associated program paths to the user for inspection.

A. Constructing CFG

Given a smart contract, the first step of sCompile is to construct the CFG. The CFG must capture all possible program paths. sCompile constructs the CFG based on the compiled EVM opcode. Note that in the compiled opcode, there is a unique location for the first instruction to be executed and there is a unique location for every function in the contract. Formally, a CFG is a tuple $(N, root, E)$ such that

- N is a set of nodes. Each node represents a basic block of opcodes (i.e., a sequence of opcode instructions which do not branch).
- $root \in N$ is the first basic block of opcodes.
- $E \subseteq N \times N$ is a set of edges. An edge (n, n') is in E if and only if there exists a control flow from n to n'.

For simplicity, we skip the details on how E is precisely defined and refer the readers to the formal semantics of EVM in [7]. In order to support inter-contract function calls, when a `CALL` instruction calls a foreign function defined in an unknown third-party contract, we assume that the foreign function may in turn call any function defined in current function[2]. Note that this assumption includes the case of calling the fallback function in third-party contract.

For instance, Fig. 3 shows the CFG of the contract *toyDAO* shown in Fig. 1. Each node in Fig. 3 represents a basic block with a name in the form of $Node\_m\_n$, where m is the index of the first opcode of the basic block and n is the index of the last. In Fig. 3, the red diamond node at the top is the $root$ node; the blue rectangle nodes represent the first node of a function. For example, $Node\_102\_109$ is the first node of function *donate()*. Note that a black oval represents a node

[1] MAIAN sends a value of 256 wei to the contract deployed in the private blockchain network

[2] The return value from a call to a foreign function is marked as symbolic during symbolic execution.

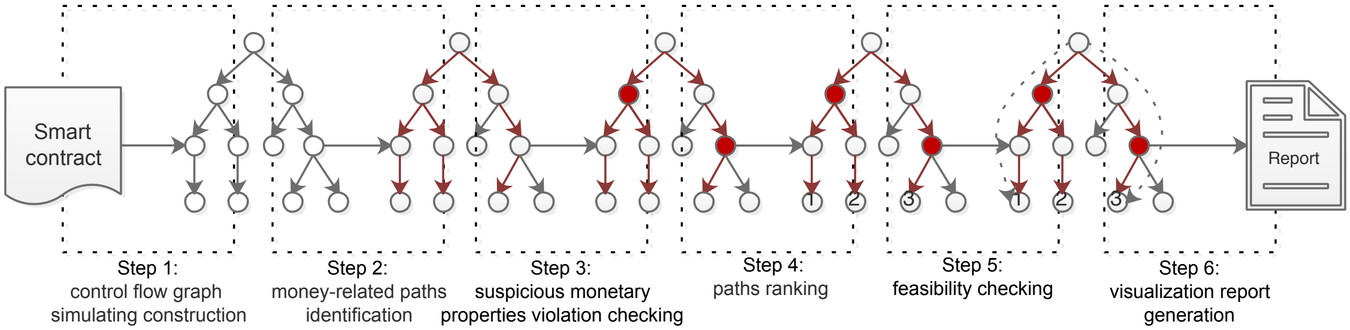

Fig. 2: Overall workflow of sCompile

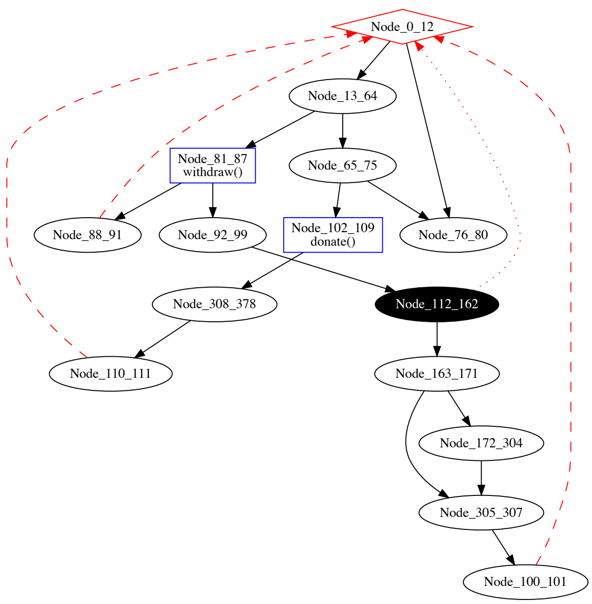

Fig. 3: Control flow graph of the contract *toyDAO*

which contains a `CALL` instruction and thus the node can be redirected to the root due to inter-contract function calls. We use different lines to represent different edges in Fig. 3. The black solid edges represent the normal control flow. The red dashed edges represent control flow due to a new function call, e.g., the edge from $Node\_88\_91$ to $Node\_0\_12$. That is, for every node n such that n ends with a terminating opcode instruction (i.e., `STOP`, `RETURN`), we introduce an edge from n to $root$. The red dotted edges represent control flow due to the inter-contract function call. That is, for every node which ends with a `CALL` instruction to an external function, an edge is added from the node to the root, e.g., the edge from $Node\_112\_162$ to $Node\_0\_12$.

Given a bound b on the number of function calls, we can systematically unfold the CFG so as to obtain all program paths during which only b or fewer functions are called. For instance, with a bound 2, the set of program paths include all of those which visit $Node\_81\_87$ or $Node\_102\_109$ no more than twice.

Statically constructing the CFG is non-trivial due to *indirect jumps* in the bytecode generated by the Solidity compiler. For instance, part of the bytecode for contract *toyDAO* is shown as follows.

```
...........                        |        .......
 92 JUMPDEST                       |        300 SHA3
 93 PUSH2 0x0064    //  100        |        301 DUP2
 96 PUSH2 0x0070    //  112        |        303 SSTORE
 99 JUMP                           |        304 POP
                                   |
100 JUMPDEST                       |        305 JUMPDEST
101 STOP                           |        306 POP
.......                            |        307 JUMP
112 JUMPDEST                       |        ........
113 PUSH1 0x00                     |
115 PUSH1 0x14                     |
.......                            |
```

The Solidity compiler applies templates to translate a Solidly program statements to EVM bytecode and often introduces indirect jumps. In the above example, The `JUMP` at line 99 is a direct jump because its target is pushed as a constant value ($0x70$) by the `PUSH2` instruction at line 96. The instruction `JUMP` at line 307 is an indirect jump because the target of `JUMP` is the top entry of the stack when execution reaches line 307. The content of the stack however cannot be determined simply by scanning the preceding instructions. In fact, the target address is pushed into the stack by the `PUSH2` instruction at line 93.

We thus use the following steps to construct CFG from EVM opcode:

1) Disassemble the bytecode to a sequence of opcode instructions.
2) Construct basic blocks from the opcode instructions (such that each basic block is a node in the CFG).
3) Connect basic blocks with edges (including but not limit to direct jumps) which can be statically decided from the opcode instructions.
4) Use stack simulation to complete the CFG with edges for indirect jumps.

In step 1, we use the disassembly utility provided by Solidity compiler to convert the bytecode to a human readable sequence of opcode instructions. In step 2, we break the opcode instructions into basic blocks such that all instructions inside a basic block execute sequentially (e.g., the basic block $Node\_92\_99$). The boundaries between basic blocks are determined by the following instructions: branching instructions `JUMP` and `JUMPI`, `JUMPDEST` which denote the start of a basic block (the entry basic block starts at address 0 which is not a `JUMPDEST` instruction), and `CALL` whose next instruction denotes a start of a new basic block, and terminal instructions such as `RETURN`, `STOP`, and `REVERT` which denote the end of a terminating basic block (e.g., the basic block $Node$_100_101).

The terminal instructions do not have a successor block and a basic block which ends with a `CALL` has two successor. One is the basic block whose first instruction is the next instruction of the `CALL` instruction in the instruction sequence. The other is the entry basic block because of the assumption about a `CALL` instruction. One successor of instruction `JUMPI` is the basic block which starts with the next instruction of the `JUMPI`.

The target address of `JUMP` and of the other successor of `JUMPI` are stored in the top stack entry when execution reaches `JUMP` and `JUMPI`. In most of the cases, the top stack entry is pushed in as a constant value by the `PUSH` instruction proceeding it (a.k.a. a direct jump). Thus the successor block can be determined statically by checking the constant value of the `PUSH` instruction.

For indirect jumps, the target of `JUMP` may be pushed by an instruction far away from the `JUMP` instruction and thus cannot be decided by checking the proceeding instructions. Thus after step 3, the basic blocks which end with an indirect jump have missing edges to their successors and we call these basic blocks as *dangling block*s (e.g., the basic block $Node$_305_307) and some basic blocks may not be reachable from the entry basic block due to the dangling blocks (e.g., the basic block $Node$_100_101).

We use *stack simulation* to find the successor of dangling basic blocks. Stack simulation is similar to define-use analysis except that dangling blocks which are reachable from the entry basic block are processed first. First, we find all the paths from the entry block to the dangling blocks (e.g., there are two paths from entry block $Node$_0_12 to the dangling block $Node$_305_307) and simulate the instructions in each path following the semantics of the instruction on the stack. Note that a dangling block ends with `JUMP` may have multiple successors in the CFG. When we reach the `JUMP` or `JUMPI` in the dangling block, the content of the top stack entry shall be determined and we connect the dangling block with the block which starts at the address as in the top stack entry. For instance, for the dangling block $Node$_305_307, there is only one successor $Node$_100_101 in both paths which is pushed by the instruction at address 093. We repeat the above step until all dangling blocks are processed.

B. Identifying Monetary Paths

Given a bound b on the number of call depth (i.e., the number of function calls) and a bound on the loop iterations, there could be still many paths in the CFG to be analyzed. For instance, there are 6 program paths in the *toyDAO* contract with a call depth bound of 1 (and a a loop bound of 5) and 1296 with a call depth bound of 4. This is known as the path explosion problem [16]. Examining every one of them, either automatically or manually, is likely infeasible. Thus, it is important that we focus on the important ones. In this work, we focus mostly on the program paths which are money-related. The reason is although there are a variety of vulnerabilities [17], almost all of them are 'money'-related as attackers often target vulnerability in the smart contracts for monetary gain.

To systematically identify money-related program paths, we label the nodes in the CFG with a flag indicating whether it is money related or not. A node is money-related if and only if its basic block contains any of the following opcode instructions: `CALL`, `CREATE`, `DELEGATECALL` or `SELFDESTRUCT`. In general, one of these opcode instructions must be used when Ether is transferred from one account to another. A program path which traverses through a money-related node is considered money-related. Note that each opcode instruction in EVM is associated with some gas consumption which technically makes them money-related. However, the gas consumption alone in most cases does not constitute vulnerabilities and therefore we do not consider them money-related.

For instance, given the CFG of *toyDAO* shown in Fig. 3, $Node$_112_162 contains a `CALL` instruction, implementing statement `msg.sender.call.value(value)()`, and thus is money-related. Any path that traverses through $Node$_112_162 is a money-related path. In Fig. 3, we visualize money-related nodes with black background.

Focusing on money-related paths allows us to reduce the number of path to analyze. For instance, the number of paths is reduced from 6 to 2 with a bound 1 and it is reduced from 1296 to 116 such paths with a bound 4.

C. Identifying Property-Violating Paths

After the previous step, we are left with a set of important program paths. To prioritize the program paths for user inspection, we proceed to analyze these paths in order to check whether critical properties are potentially violated or not. The objective is to prioritize those program paths which may trigger violation of critical properties for user inspection. In the following, we introduce some of the properties that we focus on in detail and discuss the rest briefly. We remark that these properties are designed based on previously known vulnerabilities. Furthermore, the properties can be configured and extended in sCompile.

Property: Respect the Limit In sCompile, we allow users to set a limit on the amount of Ether transferred out of the contract's wallet. This is inspired by common practice applied in banking systems. For each program path, we statically check whether Ether is transferred out of the wallet and whether the transferred amount is potentially beyond the limit. That is, for each program path which transfers Ether, we use a symbolic variable to simulate the remaining limit, which is initialized to be the limit. Each time an amount is transferred out, we decrease the variable by the same amount. Afterwards, we check whether the remaining limit (i.e., a symbolic expression) is less than zero. If it is, we conclude that the program path potentially violates the property. Note that if we are unable to determine the exact amount to be transferred (e.g., it may depend on the user input), we conservatively assume the limit may be broken.

For instance, assume that the limit is set to be 30wei for the *toyDAO* contract shown in Fig. 1, the following path is reported to exceed the transfer limit: $Node\_0\_12 \rightarrow \cdots \rightarrow Node\_112\_162 \rightarrow Node\_0\_12 \rightarrow \cdots \rightarrow Node\_112\_162$. Initially, the limit has value 30 (i.e., the assumed user-set limit). Each time $Node\_112\_162$ is executed, its value is reduced by 20. Thus, its value becomes negative after the second time.

Property: Avoid Non-Existing Addresses Any hexadecimal string of length no greater than 40 is considered a valid (well-formed) address in Ethereum. If a non-existing address is used as the receiver of a transfer, the Solidity compiler does not generate any warning and the contract can be deployed on Ethereum successfully. If a transfer to a non-existing address is executed, Ethereum automatically registers a new address (after padding 0s in front of the address so that its length becomes 160bits). Because this address is owned by nobody, no one can withdraw the Ether in it since no one has the private key.

For every program path which contains instruction `CALL` or `SELFDESTRUCT`, sCompile checks whether the address in the instruction exists or not. This is done with the help of EtherScan, which is a block explorer, search, API and analytic platform for Ethereum [13]. Given an address, EtherScan makes use of the public ledger of Ethereum, and returns true if it is registered (i.e., the address has come to effect with a cost of 25,000wei for this account, and there is at least one transaction history record). Otherwise, returns false. A program path which sends Ether to a non-existing address is considered to be violating the property. There are 2 types of transactions to register an address in Ethereum: external transactions which are initiated by an external account and internal transaction which are initiated by other contracts through function calls to the address. Most of the addresses are registered by external transactions. To minimize the number of requests to EtherScan, we only query external transactions, thus may lead to false positives when the address has only internal transactions.

For instance, in the contract *EnjinBuyer* shown in Fig. 1, address *sale* is less than 160 bits (due to omitting the last 4 bits). sCompile checks the validity of the address *sale* in a program path which calls function *purchase_tokens()* and warns the user that it is not an existing address. As a result, the user can capture such mistakes.

Property: Guard Suicide sCompile checks whether a program path would result in destructing the contract without constraints on the date or block number, or the contract ownership. A contract may be designed to "suicide" after certain date or reaching certain number of blocks, and often by transferring the Ether in the contract wallet to the owner. If however a program path which executes the opcode instruction `SELFDESTRUCT` can be executed without constraints on the date or block number, or the contract ownership, the contract can be destructed arbitrarily and the Ether in the wallet can be transferred to anyone. A famous example is Parity Wallet [18] which resulted in an estimated loss of tokens worthy of $155 million [19].

```
   contract StandardToken is Token {
1     function destroycontract(address _to) {
2        require(now > start + 10 days);
3        require(msg.sender != 0);
4        selfdestruct(_to);
5     }
6     ...
7  }
8  contract Problematic is StandardToken { ... }
```

Fig. 4: Guardless suicide

We thus check whether there exists a program path which executes `SELFDESTRUCT` and whether its path condition is constituted with constraints on date or block number and contract owner address. While checking the former is straightforward, checking the latter is achieved by checking whether the path contains constraints on instruction `TIMESTAMP` or `BLOCK`, and checking whether the path condition compares the variables representing the contract owner address with other addresses. A program path which calls `SELFDESTRUCT` without such constraints is considered a violation of the property.

One example of such vulnerability is the *Problematic* contract[3] shown in Fig. 4. Contract *Problematic* inherits contract *StandardToken*, which provides basic functionalities of a standard token. One of the functions in *StandardToken* is *destroycontract()*, which allows one to destruct the contract. sCompile reports a program path which executes line 4 potentially violates the property.

Property: Be No Black Hole In a few cases, sCompile analyzes program paths which do not contain `CALL`, `CREATE`, `DELEGATECALL` or `SELFDESTRUCT`. For instance, if a contract has no money-related paths (i.e., never sends any Ether out), sCompile then checks whether there exists a program path which allows the contract to receive Ether. The idea is to check whether the contract acts like a black hole for Ether. If it does, it is considered a vulnerability.

To check whether the contract can receive Ether, we check whether there is a *payable* function. Since Solidity version 0.4.x, a contract is allowed to receive Ether only if one of its public functions is declared with the keyword *payable*. When the Solidity compiler compiles a non-payable function, the following sequence of opcode instructions are inserted before the function body

```
CALLVALUE
```

[3]We hide the names of the contracts as some of them are yet to be fixed.

```
2  ISZERO
3  PUSH XX
4  JUMPI
5  PUSH1 0x00
6  DUP1
7  REVERT
```

At line 1, the instruction CALLVALUE retrieves the message value (to be received). Instruction ISZERO then checks if the value is zero, if it is zero, it jumps (through the JUMPI instruction at line 4) to the address which is pushed into stack by the instruction at line 3; or it goes to the block starting at line 5, which reverts the transaction (by instruction REVERT at line 7). Thus, to check whether the contract is allowed to receive Ether, we go through every program path to check whether it contains the above-mentioned sequence of instructions. If all of them do, we conclude that the contract is not allowed to receive Ether. Otherwise, it is. If the contract can receive Ether but cannot send any out, we identify the program path for receiving Ether as potentially violating the property and label it with a warning messaging stating that the contract is a black hole.

For instance, given contract *Bitway* shown in Fig. 1, the program path corresponding to a call of function *approve()* contains the following sequence of instructions.

```
0305 JUMPDEST
0306 CALLVALUE //get the msg.value
0307 ISZERO
0308 PUSH2 013c //if msg.value is 0, go to line 316
0311 JUMPI
0312 PUSH1 00
0314 DUP1
0315 REVERT
0316 JUMPDEST //start of main block
```

As a comparison, the program path corresponding to a call of function *createTokens()* does not contain the sequence of instructions. At the same time, there is no instruction like CALL, CREATE, DELEGATECALL and SELFDESTRUCT in its EVM code to send Ether out, so the contract *Bitway* is a contract which receives Ether but never sends any out.

We have presented above a few built-in properties supported by sCompile. These properties are designed based on reported vulnerabilities. sCompile is designed to be extensible, i.e., new properties can be easily supported by providing a function which takes a program path as input and reports whether the property is violated or not.

To further help users understand program paths of a smart contract, sCompile supports additional analysis. For instance, sCompile provides analysis of gas consumption of program paths. Gas is the price for executing any part of a contract. It helps to defend against network abuse as execution of any EVM bytecode instruction consumes a certain amount of gas. To execute a normal transaction successfully, enough gas must be provided; otherwise the transaction will fail and the consumed gas will be forfeited. For every transaction, Ethereum estimates the amount of gas to be consumed by the transaction based on concrete transaction inputs provided by the user. However, without trying out all possible inputs, users of the contract may not be aware of the existence of certain particularly gas consuming program paths. Given a contract, sCompile estimates the gas consumption of all program paths found by symbolic execution and then output the maximum gas consumption of corresponding path(s). The gas consumption of a program path is estimated based on each opcode instruction in the program path statically.

TABLE I: Definition of α_{pr}

	transfer limit	non-existing addr.	suicide	black hole
Likelihood	1	1	2	3
Severity	2	3	3	2
Difficulty	2	2	3	2
α_{pr}	4	6	18	12

D. Ranking Program Paths

So far we have identified a number of program paths, some of which potentially violates certain properties. To allow user to focus on the most critical program paths as well as to save the effort on applying heavy analysis techniques like symbolic execution on these program paths, we prioritize the program paths according to the likelihood they reveal critical vulnerability in the contract. For each program path, we calculate a criticalness score and then rank the program paths according to the scores. The criticalness score is calculated using the following formula: let pa be a program path and V be the set of properties which p violates.

$$criticalness(pa) = \frac{\Sigma_{pr \in V} \alpha_{pr}}{\epsilon * length(pa)}$$

where α_{pr} is a constant which denotes the criticalness of violating property pr, $length(pa)$ is the length of path pa (i.e., the number of function calls) and ϵ is a positive constant. Intuitively, the criticalness is designed such that the more critical a property the program path violates, the larger the score is; and the more properties it violates, the larger the score is. Furthermore, it penalizes long program paths so that short program paths are presented first for user inspection. Note that a program path may violate multiple properties. For instance, a path which transfers all Ethers to an non-existing account before destructing the contract violates property of non-existing address as well as property on guardless suicide.

To assess the criticalness of each property, we use the technique called failure mode and effects analysis (FMEA [20]) which is a risk management tool widely used in a variety of industries. FMEA evaluates each property with 3 factors, i.e., *Likelihood*, *Severity* and *Difficulty*. Each factor is a value rating from 1 to 3, i.e., 3 for *Likelihood* means the most likely; 3 for *Severity* means the most severe and 3 for *Difficulty* means the most difficult to detect. The criticalness α_{pr} is then set as the product of the three factors. After ranking the program paths according to their criticalness score, only program paths which have a criticalness score more than certain threshold are subject to further analysis. This allows us to reduce the program paths significantly.

In order to identify the threshold for criticalness, we adapt the k-fold cross-validation[21], [22] idea in statistical area.

```
  contract GigsToken {
1   function createTokens() payable {
2     require(msg.value > 0);
3     uint256 tokens = msg.value.mul(RATE);
4     balances[msg.sender] = balances[msg.sender].add(tokens);
5     owner.transfer(msg.value);
6   }
7   ...
  }
```

Fig. 5: A non-greedy contract

We collected a large set of smart contracts and split them into a training data set(10,452 contracts) and a test data set (25,678 contracts). The training data set is used to tune the parameters required for computing the criticalness, e.g., value of ϵ and the threshold for criticalness score. We repeated the experiments 20 times which took more than 5,700 total hours of all machines and optimizes those parameters (based on the number of vulnerability discovered and the false positive rate of each property). The parameters adapted for each property as shown in Table I, and ϵ is set to be 1 and the threshold for criticalness is set to be 10.

E. Feasibility Checking

After the last step, we have identified a ranked list of highly critical program paths which potentially reveal vulnerability in the smart contract. Not all the program paths are however feasible. To avoid false alarms, in this step, we filter infeasible program paths through symbolic execution.

Symbolic execution [23], [24] is a well-established method for program analysis. It has been applied to solve a number of software engineering tasks. The basic idea is to symbolically execute a given program, e.g., use symbolic variables instead of concrete values to represent the program inputs and maintain the constraints that a program path must satisfy in order to traverse along the path, and lastly solve the constraint using a constraint solver in order to check whether the program path is feasible or not. Symbolic execution has been previously applied to Solidity programs in Oyente [9] and MAIAN [14]. In this work, we apply symbolic execution to reduce the program paths which are to be presented for users' inspection. Only if a program path is found to be infeasible by symbolic execution, we remove it. In comparison, both Oyente and MAIAN aim to fully automatically analyze smart contracts and thus when a program path cannot be determined by symbolic execution, the result may be a false positive or negative.

For instance, Fig. 5 shows a contract which is capable of both receiving (since the function is *payable*) and sending Ether (due to *owner.transfer(msg.value)* at line 5), and thus sCompile does not flag it to be a black hole contract. MAIAN however claims that it is. A closer investigation reveals that because MAIAN has trouble in solving the path condition for reaching line 5, and thus mistakenly assumes that the path is infeasible. As a result, it believes that there is no way Ethers can be sent out and thus the contract is a black hole.

F. User Inspection

The last step of our approach is to present the analysis results for user inspection. For user's convenience, we implemented a graphical user interface (GUI) in sCompile. The GUI is not limited to display the final analysis results. For the first step, user could open a smart contract in GUI by either open or copy/paste the source code of smart contract. User has the options to customize various parameters used in the analysis, i.e., the bound on the call depth, the transfer limit, the bound on loop iteration, the threshold for the criticalness and the criticalness of various properties. After the analysis, the output to the user consists of mainly two parts, i.e., one part on statistical data and the other on detailed path data. For the statistical data, a report is displayed which shows the total execution time, the number of symbolic analyzed path and the number of warnings for each properties that are discussed in section III-C. For the path data, the top ranked critical paths (which have a criticalness more than the threshold and are not proved infeasible by symbolic execution) are shown to user in the form of function call sequences. That is, for each critical program path, we map it back to a sequence of function calls by identifying the basic blocks in the sequence which represent the start of a function. Furthermore, if the constraint solver is able to solve the path condition, concrete function parameters are used. Each critical path is associated with a warning message which explains why the program path should be inspected by the user. e.g., a potential violation of a critical property or being particularly gas-consuming. User can click on a specific path and the part of code which are associated with the path is highlighted in the source code.

IV. Implementation and Evaluation

A. Implementation

sCompile is implemented in C++ with about 8K lines of code. The source code is available online[4]. The symbolic execution engine in sCompile is built based on the Z3 SMT solver [25]. Note that we also symbolically execute the constructor in the contract and use the resultant symbolic states as the initial states for symbolic execution of all other functions in the contract.

B. Experiment

In the following, we evaluate sCompile to answer research questions (RQ) regarding sCompile's efficiency, effectiveness and usefulness in practice. Our test subjects contain all 36,099 contracts (including both the training set and the set) with Solidity source code which are downloaded from EtherScan. Although sCompile can also take EVM code as input, we apply sCompile to Solidity source code so that we can manually inspect the experiment results.

All experiment results reported below are obtained on a machine running on Amazon EC2 C3 xlarge instance type with Ubuntu 16.04 and gcc version 5.4.0. The detailed hardware configuration is: 2.8 GHz Intel Xeon E5-2680 v2

[4] The link is removed for anonymity.

TABLE II: Execution time of sCompile vs. Oyente vs. MAIAN

	sCompile (call bound = 1	sCompile (call bound = 2)	sCompile (call bound = 3)	Oyente	MAIAN (Suicidal)	MAIAN (Prodigal)	MAIAN (Greedy)
median	3.106	8.717	5.267	18.015	19.053	23.472	19.397
#timeout	1145	1737	2597	2223	1561	6186	1081

TABLE III: Loop bound definitions among three tools

Tool	call bound	loop bound	timeout	other bound
sCompile	3	5	60 s	60 cfg nodes
Oyente	1	10	60 s	N.A.
MAIAN	3 (no inter-contract)	N.A.	60 s	60 cfg nodes

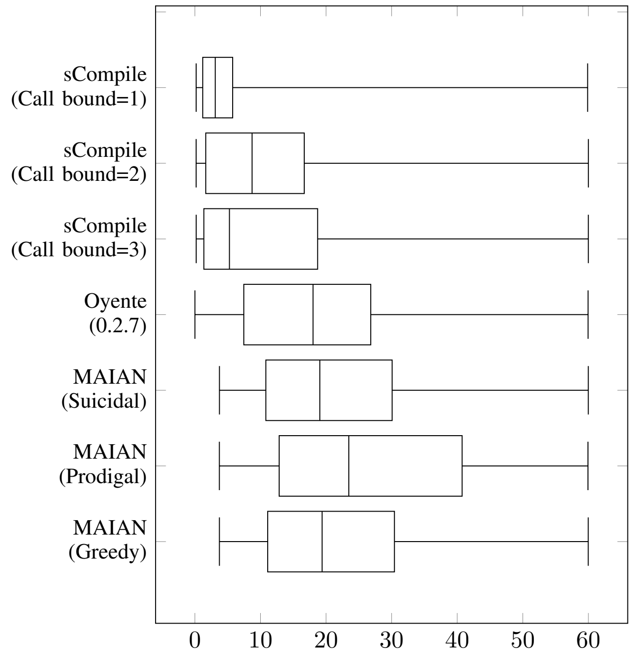

Fig. 6: Execution time of sCompile vs. Oyente vs. MAIAN

processor, 7.5 GB ram, 2 x 40 GB SSD. The timeout set for sCompile is: global wall time is 60 seconds and Z3 solver timeout is 100 milliseconds. Furthermore, the limit on the maximum number of blocks for a single path is set to be 60, and the limit on the maximum iterations of loops is set to be 5, i.e., each loop is unfolded at most five times.

RQ1: Is sCompile efficient enough for practical usage? sCompile is designed to be an addon toolkit for Solidity compiler and thus it is important that sCompile is able to provide timely feedback to users when a smart contract is implemented and compiled. In this experiment, we evaluate sCompile in terms of its execution time. We systematically apply sCompile to all the benchmark programs in the training set (which includes all the contracts in EtherScan as of January 2018) and measure the execution time (including all steps in our approach).

The results are summarized in Table II and Fig. 6. In Table II, the second, third and fourth column show the execution of sCompile with call depth bound 1, 2, and 3 respectively, so that we observe the effect of different call depth bounds. For baseline comparison, the fifth column shows the execution time of Oyente (the latest version 0.2.7) with the same timeout. We remark that the comparison should be taken with a grain of salt. Oyente does not consider sequences of function calls, i.e., its bound on function calls is 1. Furthermore, it does not consider initialization of variables in the constructor (or in the contract itself). The next columns show the execution time of MAIAN (the latest commit version on Mar 19). Although MAIAN is designed to analyze program paths with multiple (by default, 3) function calls, it does not consider the possibility of a third-party contract calling any function in the contract through inter-contract function calls and thus often explores much fewer program paths than sCompile. Furthermore, MAIAN checks only one of the three properties (i.e., suicidal, prodigal and greedy) each time. Thus, we must run MAIAN three times to check all three properties. The different bounds used in all three tools are summarized in Table III.

In Table II, the second row shows the median execution time and the third row shows the number of times the execution time exceeds the global wall time (60 seconds). We observe that sCompile almost always finishes its analysis within 10 second. Furthermore, the execution time remains similar with different call depth bounds. This is largely due to sCompile's strategy on applying symbolic execution only to a small number of top ranked critical program paths. We do however observe that the number of timeouts increases with an increased call depth bound. A close investigation shows that this is mainly because the number of program paths extracted from CFG is much larger and it takes more time to extract all paths for ranking. In comparison, although Oyente has a call depth bound of 1, it times out on more contracts and spends more time on average. MAIAN spends more time on each property than the total execution of sCompile. For some property (such as *Greedy*), MAIAN times out fewer times, which is mainly because it does not consider inter-contract function calls and thus works with a smaller CFG.

Fig. 6 visualizes the distribution of execution time of the tools. The horizontal-axis represents the execution time (in seconds). Five numbers are used to construct each row: min, lower quartile, median, upper quartile and max. The leftmost and rightmost vertical lines represents the minimal and the maximum respectively. For each box, The leftmost and rightmost vertical line represents the lower quartile (i.e., the median of the lower half $1/4$) and upper quartile value (i.e., the median of the upper half $3/4$). The middle vertical line inside the box is the median value. Based on the data, we conclude that sCompile is efficient.

TABLE IV: Comparison on vulnerable contracts

	sCompile			MAIAN		
	alarmed	true positive	false positive	alarmed	true positive	false positive
Avoid non-existing address	37	32	5	N.A.	N.A.	N.A.
Be no black hole	57	57	0	141	56	85
Guard suicide	42	38	4	66	30	36

TABLE V: Average number of program paths

	in total	symbolic-executed	to user
call depth 1	48.92	37.51	1.49
call depth 2	6177.21	144.24	12.46
call depth 3	31346.62	121.23	12.62

We conjecture that the main reason that sCompile can efficiently analyze smart contracts is that heavy techniques like symbolic execution are applied only to the most critical program paths in sCompile. To validate the conjecture, we count the average number of program paths which are analyzed through symbolic execution in sCompile. Table V shows the results. The second column shows the estimated total number of program paths on average for each smart contract which is successfully analyzed. Note that the estimation is based on the CFG and thus may count program paths which are infeasible. This is part of the reason it is often greater than the number reported by alternative methods like [9], [14]. The other part is that our CFG is more complete. The third column shows the average number of paths analyzed with symbolic execution. It can be observed that only a small fraction of the program paths are symbolically analyzed. Furthermore, the number of symbolically executed paths remain small even when the call depth bound is increased. This is because only the top ranked critical program paths are analyzed by symbolic execution. If there are multiple program paths which potentially violate the same property, sCompile prioritizes the shorter one and often avoids symbolically executing the longer one. The results confirm our conjecture.

In the second experiment, we aim to investigate the effectiveness of sCompile. We apply sCompile to all 36,099 contracts and manually inspect the critical paths reported by sCompile to check whether the program path, together with the associated warning message, reveals a true vulnerability in the contract. Note that not all properties checked by sCompile readily signals a vulnerability. For instance, given a user-set transfer limit, sCompile may report that a program path violates the transfer limit. Although such information is often useful, depending on the transfer limit set by the user, the program path may or may not signal a vulnerability. For instance, a gambling contract may allow a user to place a bet with certain amount and transfer some amount back to the user when the betting result is revealed. In such a case, the transfer limit is likely broken if a large bet is placed by the user. For another instance, sCompile automatically reports a program path which is the most gas-consuming. Such information is useful for the user (e.g., to set the right 'price' for the transaction). It however does not necessarily signal a vulnerability (although it may signal program bugs). We thus focus on those results produced by sCompile which are directly related to vulnerabilities in the following, i.e., program paths which are deemed to violate property "avoid non-existing addresses", "be no black hole" and "guard suicide". Note that two of the properties (i.e., the latter two) analyzed by sCompile are supported by MAIAN as well. We can thus compare sCompile's performance with that of MAIAN for these two properties. The results are shown in Table IV. In the following, we discuss the detailed findings[5].

For *Property: Be no Black Hole*, there are 57 contracts in the training set are marked vulnerable by sCompile. We manually checked all these contracts and confirm that they are all true positives. In comparison, MAIAN identified 141 black hole contracts and 56 contracts among them are true positives, 43 of which overlap with sCompile's results. We then investigate why sCompile missed the remaining 13 contracts identified by MAIAN. We discovered all of them took more than 60 seconds and thus sCompile timed out before finishing analyzing. When we set the timeout to 200s, sCompile identifies 3 more as black hole contracts.

The other 85 identified by MAIAN are false positives. Our investigation reveals that 62 of them are library contracts. Because MAIAN does not differentiate library contracts from normal contracts, it marks all library contracts as vulnerable. We randomly choose 5 contracts from the remaining for further investigation. We find Z3 could not finish solving the path condition in time and thus MAIAN conservatively marks the contract as vulnerable. After extending the time limit for Z3 and the total timeout, 4 of the 5 false positives are still reported. The reason is that these contracts can only send Ether out after certain period, and MAIAN could not find a feasible path to send Ether out for such cases, and mistakenly flags the contract as a black hole.

For *Property: Guard Suicide*, sCompile reports a program path if it leads to `SELFDESTRUCT`, without a constraint on the ownership of the contract or the date or the block number, i.e., a guard to prevent an unauthorized users from killing the contract. Among the analyzed contracts, sCompile identified 42 contracts which contain at least one program path which violates the property. Many of the identified contracts violate the property due to contract inheritance as shown in Fig. 4.

[5]We have informed all developers whose contact info are available about the vulnerabilities in their contracts and several have confirmed the vulnerabilities and deployed new contracts to substitute the vulnerable ones. Some are yet to respond, although the balance in their contracts are typically small.

```
   contract ViewTokenMintage{
1    modifier auth {
2        require(isAuthorized(msg.sender, msg.sig));
3        _;
4    }
5    function isAuthorized(address src, bytes4 sig)
        internal view returns (bool) {
6        if (src == address(this)) {
7            return true;
8        } else if (src == owner) {
9            return true;
10        } else if (authority == DSAuthority(0)) {
11            return false;
12        } else {
13            return authority.canCall(src, this, sig);
14        }
15    }
16    function destruct(address addr) public auth {
17        selfdestruct(addr);
18    }
   }
```

Fig. 7: False positive on guardless suicide by sCompile

```
contract MiCarsToken {
    function killContract () payable external {
      if (msg.sender==owner ||
              msg.value >=howManyEtherInWeiToKillContract)
          selfdestruct(owner);
    }
    ...
}
```

Fig. 8: Ambiguous cases between sCompile and MAIAN

The remaining 4 cases reported by sCompile are false positives. We manually investigate them one by one. We find in one case, the contract is set up such that only the sender of original transaction can trigger SELFDESTRUCT. This is rather uncommon way of coding. The other 3 false alarms are from the same contract *ViewTokenMintage* shown in Fig. 7. The guard of *selfdestruct* depends on the return value of function *isAuthorized()*. The path going through line 6 returns true only if the *msg.sender* is the same as the current contract. sCompile mistakenly reports the alarm as the *ADDRESS* is symbolized as a symbolic constant.

Different from sCompile, MAIAN only checks whether a contract can be destructed without any constraints except an ownership constraint. MAIAN identified 66 contracts violating the property. 30 of them are true positives, 13 of which are also identified by sCompile. The other 36 are false positives. The contract *MiCarsToken* shown in Fig. 8 shows a typical false alarm. There are 2 constraints before SELFDESTRUCT in the contract. sCompile considers such a contract safe for there is a guard of *msg.sender* == *owner* (or the other condition), whereas MAIAN reports a vulnerability as the contract can also be killed if the msg.sender is not the owner when the second condition is satisfied.

We further analyzed the 17 cases which were neglected by sCompile. 6 of them are alarmed for owner change as exemplified in Fig. 9. In this contract, *selfdestruct* is well guarded, but the developer makes a mistake so that the constructor becomes a normal function, and anyone can invoke *mortal()* to make himself the owner of this contract and kill the contract.

```
contract Mortal {
    address public owner;
    function mortal() { owner = msg.sender; }
    function kill() {
      if (msg.sender == owner) suicide(owner); }
}
```

Fig. 9: Contract of owner change

For *Property: Avoid Non-existing Address*, as demonstrated by contract EnjinBuyer in Fig. 1, it is a problem if a wrong address is used. For the contracts in the training set, all addresses identified are of length 160 bits. However, there are 37 contracts identified as non-existing addresses (i.e., not registered in Ethereum mainnet). These non-existing addresses may be used for different reasons. For example, in contract *AmbrosusSale*, the address of TREASURY does not exist before the function `specialPurchase()` or `processPurchase()` is invoked. As a result, it costs more gas for a user who is the first to invoke those 2 functions because account registration costs at least additional $25,000$wei, which the user may not be aware of. There are 5 addresses from 5 contracts which are registered by internal transactions.

We further analyzed 25,647 contracts newly uploaded in EtherScan from February 2018 to July 2018. For *"Be no Black Hole"*, there are 109 vulnerabilities out of 139 alarms generated by sCompile. Applying MAIAN on these contracts, 84 of them are marked vulnerable, 77 of which are true vulnerabilities overlapping with those found by sCompile and 7 library contracts are marked vulnerable mistakenly. Among the 139 contracts, 25 vulnerable ones are missed by MAIAN according to our manual check. For *"Guard Suicide"*, there are 83 vulnerabilities out of 114 alarms generated by sCompile. Applying MAIAN on these contracts, 42 are marked vulnerable, all of which overlap with those found by sCompile. For *"Avoid Non-existing Addresses"*, there are 80 vulnerabilities out of 87 alarms generated by sCompile. The 7 false alarms are due to internal transactions.

In total, sCompile identifies 224 new vulnerabilities from the 36,099 contracts consisting of 46 *Black Hole* vulnerabilities, 66 *Guardless Suicide* vulnerabilities and 112 *Non-existing Address* vulnerabilities.

RQ3: Is sCompile useful to contract users? Different from other tools which aim to fully automatically analyze smart contracts, sCompile is designed to facilitate human users. We thus conduct a user study to see whether sCompile is helpful for users to detect vulnerabilities.

The user study takes the form of an online test. Once a user starts the test, first the user is briefed with necessary

TABLE VI: Statistics and results of surveyed contracts

Contract	LOC	#paths	Q1	Q2	Time	Usefulness
C1 (w)	33	8	7/8	3/8	119	
C2 (w)	52	16	7/8	2/8	98	
C3 (w)	67	38	7/8	2/8	233	5
C4 (w/o)	87	59	2/8	1/8	414	
C5 (w/o)	103	13	3/8	1/8	397	
C6 (w/o)	107	27	4/8	1/8	420	

background on smart contract vulnerabilities (with examples). Then, 6 smart contracts (selected at random each time from a pool of contracts) are displayed one by one. For each contract, the source code is first shown. Afterwards, the user is asked to analyze the contract and answer the two questions. The first question asks what is the vulnerability that the contract has. The second question requires user to identify the most gas consuming path in the contract (with one function call).

For the first three contracts, the outputs from sCompile are shown alongside the contract source code as a hint to the user. For the remaining 3 contracts, the hints are not shown. The contracts are randomized so that not the same contracts are always displayed with the hint. The goal is to check whether users can identify the vulnerabilities correctly and more efficiently with sCompile's results.

We distribute the test through social networks and online professional forums. We also distribute it through personal contacts who we know have some experience with Solidity smart contracts. In three weeks we collected 48 successful responses to the contracts (without junk answers)[6]. Table VI summarizes the results. Recall that sCompile's results are presented for the first three contracts. Column LOC and #paths shows the number of lines and program paths in each contract. Note that in order to keep the test manageable, we are limited to relatively small contracts in this study. Columns Q1 and Q2 show the number of correct responses (the numerator) out of the number of valid responses (the denominator). We collect the time (in seconds) taken by each user in the Time column to answer all the questions. In the end of the survey we ask the user to give us a score (on the scale of 1 to 7, the higher the score the more useful our tool is) on how useful the hints in helping them answer the questions. The value in column Usefulness is the average score over all responses because all responses are shown half the hints.

The results show that for the first three contracts for which sCompile's analysis results are shown, almost all users are able to answer Q1 correctly using less time. For the last three contracts without the hints, most of the users cannot identify the vulnerability correctly and it takes more time for them to answer the question. For identifying the most gas-consuming path, even with the hints on which function takes the most gas, most of the users find it difficult in answering the question, although with sCompile's help, more users are able to answer the question correctly. The results show that gas consumption is not a well-understood problem and highlight the necessity of reporting the condition under which maximum gas consumption happens. All the users think our tool is useful (average score is $5/7$) in helping them identify the problems.

[6]There are about 80 people who tried the test. Most of the respondents however leave the test after the first question, which perhaps evidences the difficulty in analyzing smart contracts.

V. Related work

sCompile is related to work on identifying vulnerabilities in smart contracts. Existing work can be roughly categorized into 3 groups according to the level at which the vulnerability resides at: Solidity-level, EVM-level, and blockchain-level [17], [26]. In addition, existing work can be categorized according to the techniques they employ to find vulnerabilities: symbolic execution [9], [14], [27], [28], [29], static-analysis based approaches [30] and formal verification [10], [11]. Our approach works at the EVM-level and is based on static analysis and symbolic execution, and is thus closely related to the following work.

Oyente [9] is the first tool to apply symbolic execution to find potential security bugs in smart contracts. Oyente formulates the security bugs as intra-procedural properties and uses symbolic execution to check these properties. Among 19,366 existing Ethereum contracts, Oyente flags $8,833$ of them as vulnerable, including the vulnerability responsible for the DAO attack. However, Oyente does not perform inter-procedural analyses to check inter-procedural or trace properties as did in sCompile.

MAIAN [14] is recently developed to find three types of problematic contracts in the wild: prodigal, greedy and suicidal. It formulates the three types of problems as inter-procedural properties and performs bounded inter-procedural symbolic execution. It also builds a private testnet to valid whether the contracts found by it are true positives by executing the contracts with data generated by symbolic execution. In the high-level, both MAIAN and sCompile perform inter-procedural symbolic analyses and check the suicidal and greedy contracts. However, sCompile differs from MAIAN in the following aspects. First, sCompile makes a much more conservative assumption about a call to third-party contract which we assume can call back a function in current contract. sCompile is designed to reduce user effort rather than to analyze smart contracts fully automatically. Therefore sCompile focuses on ranking program paths in terms of their criticalness and only applies symbolic execution to selected few critical program paths. Secondly, sCompile supports more properties than MAIAN. Thirdly, sCompile checks properties in ways which are different from MAIAN. For instance, to check for black hole contracts, MAIAN checks whether a contract can receive Ether through testing (e.g., by sending Ether to the contract). As showed in Section II, the result is that there may be false negatives. Other symbolic execution based tools [27], [28] perform intra-procedural symbolic analysis directly on the EVM bytecode as what Oyente does.

The tool Securify [30] is based on static analysis to analyze contracts. It infers semantic information about control dependencies and data dependencies from the CFG of an intermediate language for EVM bytecode. Then it specifies both compliance and violation patterns for the property. The vulnerability detection problem is then reduced to search the patterns on the inferred data and control dependencies information. The use of compliance pattern reduces the number of false positives in the reported warnings. Our approach does not infer semantic information from CFG, instead in the ranking algorithm, we rely on syntactic information to reduce paths for further symbolic analysis to improve performance. We analyze the extracted paths with symbolic execution which is more precise than the pure static analysis as adopted by Securify.

In addition, static analysis based tools such as those provided in Solidity compiler and Remix IDE [31] can perform checks on the Solidity source code to find common programming anti-patterns and cannot find the properties proposed in this work.

Besides symbolic execution, there are attempts on formal verification of smart contracts using either model-checking techniques [10] or theorem-proving approaches [11]. These approaches in theory can check arbitrary properties specified manually in a form accepted by the model checker or the theorem prover. It is known that model checking has limited scalability whereas theorem proving requires an overwhelming amount of user effort.

VI. Conclusion

In this work, we introduce an approach to reveal "money-related" vulnerabilities in smart contract by identifying a small number of critical paths for user inspection. The critical paths are identified and ranked so that the effort required on applying symbolic execution techniques or user inspection is minimized. We implemented the approach in the tool sCompile. We show that sCompile can effectively and efficiently analyze smart contracts. In addition, with sCompile, we find 224 new vulnerabilities. All the new vulnerabilities are well defined in our approach and could be presented to the user in well-organized information within a reasonable time frame. In the future, we plan to further develop sCompile to improve its efficiency and effectiveness (with techniques like loop-invariant synthesis).

References

[1] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006.

[2] Whitfield Diffie and Martin E. Hellman. Multiuser cryptographic techniques. In *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, AFIPS '76, pages 109–112, New York, NY, USA, 1976. ACM.

[3] Norman D Jorstad and TS Landgrave. Cryptographic algorithm metrics. In *20th National Information Systems Security Conference*, 1997.

[4] Stuart Haber and W Scott Stornetta. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*, pages 437–455. Springer, 1990.

[5] Jerry Brito and Andrea Castillo. *Bitcoin: A primer for policymakers*. Mercatus Center at George Mason University, 2013.

[6] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[7] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[8] Bitcoin, ethereum, and litecoin price - coinbase. https://www.coinbase.com/charts. (Accessed on 12/01/2017).

[9] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[10] Sukit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharmar. Zeus: Analyzing safety of smart contracts. In *Network and Distributed Systems Security Symposium 2018*, pages 1–12. internetsociety, 2018.

[11] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96. ACM, 2016.

[12] Alan M Turing. On computable numbers, with an application to the *Entscheidungsproblem. Proc. London Math. Soc*, 42, 1937.

[13] Ethereum (eth) blockchain explorer. https://etherscan.io/. (Accessed on 06/30/2018).

[14] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *arXiv preprint arXiv:1802.06038*, 2018.

[15] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[16] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381. Springer, 2008.

[17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[18] anyone can kill your contract · issue #6995 · paritytech/parity. https://github.com/paritytech/parity/issues/6995#issuecomment-342409816. (Accessed on 06/06/2018).

[19] Another parity wallet hack explained – sergey petrov – medium. https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c. (Accessed on 06/06/2018).

[20] D. H. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution*. ASQ Quality Press, 2003.

[21] Pierre A Devijver and Josef Kittler. *Pattern recognition: A statistical approach*. Prentice hall, 1982.

[22] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[23] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[24] William E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, (4):266–278, 1977.

[25] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[26] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 2017.

[27] ConsenSys. Mythril: Security analysis of ethereum smart contracts. https://github.com/ConsenSys/mythril, 2018. [online, accessed 30-may-2018].

[28] trailofbits. Manticore: Symbolic execution tool. https://github.com/trailofbits/manticore, 2018. [online, accessed 30-may-2018].

[29] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. *arXiv preprint arXiv:1807.03932*, 2018.

[30] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. *arXiv preprint arXiv:1806.01143*, 2018.

[31] Remix-solidity ide. https://remix.ethereum.org. (Accessed on 08/01/2018).