Recommended Paper

RA: A Static Analysis Tool for Analyzing Re-Entrancy Attacks in Ethereum Smart Contracts

Yuichiro Chinen[1,a)]   Naoto Yanai[1,b)]   Jason Paul Cruz[1,c)]   Shingo Okamura[2,d)]

Abstract: Ethereum smart contracts are programs that are deployed and executed in a consensus-based blockchain managed by a peer-to-peer network. Several re-entrancy attacks that aim to steal Ether, the cryptocurrency used in Ethereum, stored in deployed smart contracts have been found in the recent years. A countermeasure to such attacks is based on dynamic analysis that executes the smart contracts themselves, but it requires the spending of Ether and knowledge of attack patterns for analysis in advance. In this paper, we present a static analysis tool named *RA* (*Re-entrancy Analyzer*), a combination of symbolic execution and equivalence checking by a satisfiability modulo theories solver to analyze vulnerability of smart contracts to re-entrancy attacks. In contrast to existing tools, RA supports analysis of inter-contract behaviors by using only the Ethereum Virtual Machine bytecodes of target smart contracts, i.e., even without prior knowledge of attack patterns and without spending Ether. Furthermore, RA can verify existence of vulnerability to re-entrancy attacks without execution of smart contracts and it does not provide false positives and false negatives. We also present an implementation of RA to evaluate its performance in analyzing the vulnerability of deployed smart contracts to re-entrancy attacks and show that RA can precisely determine which smart contracts are vulnerable.

Keywords: ethereum, smart contracts, static analysis, EVM, symbolic execution, SMT solver

1. Introduction

1.1 Backgrounds

Ethereum, which is often described as "the world computer," is a global, open-source platform for decentralized applications and execution of programs called *smart contracts*, which are software programs recorded on the Ethereum blockchain [*1] and executed in the *Ethereum Virtual Machine* (*EVM*). The EVM is a virtual machine that runs codes called *EVM bytecodes* and is the runtime environment for smart contracts in Ethereum. Ethereum enables developers to build smart contracts with built-in functions and gain the benefits of cryptocurrency and blockchain technologies.

Research Motivation: The blockchain is decentralized and transparent by nature, and thus anyone can read the bytecodes of deployed contracts. Moreover, smart contracts typically contain financially valuable data and therefore create a criminogenic environment for adversaries. The attack on "The DAO" on June 2016 is the most infamous case of an attack on smart contracts. In the attack, a vulnerability called *re-entrancy*, where the main contract calls an external contract which again calls into the calling contract within a single transaction, was utilized to steal more than 60 million US Dollars worth of Ether. With the transparency of the Ethereum blockchain, the vulnerabilities of deployed contracts can be utilized permanently as a springboard of attacks. Indeed, many attacks [5] and honeypots [32] on Ethereum have been found in the past years.

Security Analysis of Smart Contracts: Based on these backgrounds, program analysis for the security of Ethereum smart contracts is an urgent and significant research theme. In this paper, we focus on the analysis of EVM bytecodes of smart contracts instead of their corresponding source codes. Analysis of bytecodes brings a couple of advantages. First, the analysis of EVM bytecodes is independent of a high-level language that is periodically updated. For instance, the Solidity language [2], which is one of the major languages used for writing Ethereum smart contracts, is often updated and offers different capabilities for different versions. Consequently, a tool that was developed today for the analysis of contracts written in Solidity before they are deployed may not be applicable for analysis of contracts written in future versions of Solidity. In fact, deployed contracts have different versions of Solidity depending on their application and on the features of the Solidity language they require or use. Therefore, the analysis of EVM bytecodes is more desirable than the analysis of Solidity and other high-level languages. Second, the analysis of a high-level language, such as Solidity, requires an analyst to input a copy of the actual source codes of the analy-

1   Osaka University, Suita, Osaka 565–0871, Japan
2   National Institute of Technology, Nara College, Yamatokoriyama, Nara 639–1080, Japan
a)   t-yuichiro@ist.osaka-u.ac.jp
b)   yanai@ist.osaka-u.ac.jp
c)   cruz@ist.osaka-u.ac.jp
d)   okamura@info.nara-k.ac.jp

*1   Hereafter, "contract" and "smart contract" are used interchangeably but have the same meaning. Likewise, "the blockchain" will be used to refer to the Ethereum blockchain, unless otherwise specified.

sis targets. If a developer will only analyze the Solidity source codes he/she develops, then a tool that analyzes Solidity source codes will suffice. However, such a tool cannot be used if a user wants to analyze other deployed contracts but he/she does not have the corresponding source codes. In contrast, users can obtain the EVM bytecodes of deployed contracts directly from a blockchain explorer even if the corresponding source codes are unpublished and analyze their vulnerabilities anytime. Thus, using static analysis of EVM bytecodes, analysts can easily judge whether a deployed contract has benign or malicious codes, e.g., vulnerable or honeypots.

In early literature on analysis of EVM bytecodes, *formal verification* [6], [16], [18], [19], [33] is a leading approach for verifying a specification of bytecodes and *symbolic execution* [8], [21], [23], [26], [31] is used for exploring bytecodes in a depth-first search fashion by extracting control flow graphs (CFGs). According to Weiss et al. [35], rigid formal verification is often dismissed as it puts too high demands on analysts who are not experts in formal verification. On the other hand, symbolic execution can potentially support analysis of vulnerabilities via extraction of CFGs. Several prior works [23], [26], [31] succeeded in analyzing some vulnerabilities, such as to re-entrancy attacks. However, symbolic execution only outputs CFGs from programs to be analyzed and does not support detection of the vulnerabilities themselves. Accordingly, analysis becomes often heuristic if an infeasible path exists in the programs. In particular, analysis results may produce many false positives and false negatives because of an infeasible path [7].

A recent work [29] has found re-entrancy attacks that undermine existing analysis tools [19], [23], [33] by creating a new contract or calling a different function via an external contract. Although a monitoring tool was also proposed in the same paper as a countermeasure against those attacks, the tool performs monitoring based on *dynamic analysis* and is therefore unable to detect vulnerabilities unless the attacks occur during program execution. Accordingly, analysts need to implement and execute attack patterns by themselves in advance to check for vulnerabilities. Although generic attack patterns [12] have been presented formally, dynamic analysis remains impractical for analysis of smart contracts because it often requires analysts to know how an attack is launched as well as to spend Ether for the execution of contracts. Ideally, contracts should be analyzed based on only their bytecodes, i.e., with the use of *static analysis*.

Research Goal: This paper aims to *design an inter-contract static analysis tool that (1) uses only EVM bytecodes as input, (2) eliminates false negatives and false positives, and (3) does not require analysts to have a priori knowledge of the attacks on contracts.* For this goal, we focus on analysis of re-entrancy attacks [29]. As evident in the attack on "The DAO," re-entrancy attacks have a significant impact and many contracts have been found to be vulnerable to these attacks [19], [23]. Considering findings on new attacks [29] in the recent years, our research goal can prove to be important and useful. Moreover, the design of a static analysis tool that is effective against new attacks is an open challenge [29].

This paper is the full version of our previous work [10] which was presented at IEEE Blockchain 2020. In the previous work, we presented an analysis tool named RA and evaluated its performance. In this paper, we evaluate the performance of RA in a theoretical fashion. We also include additional related works from literature.

1.2 Contributions

In this paper, we present a new static analysis tool named *Re-entrancy Analyzer* (*RA*) for analyzing EVM bytecodes of smart contracts. RA can analyze re-entrancy vulnerabilities via inter-contract flows, for instance, by creating a new contract and calling a different function in the main contract via an external contract. RA does not require analysts to have prior knowledge of the attack patterns and pay Ether for analysis, and it does not provide false positives and false negatives. These advantages are achieved via integration of symbolic execution and equivalence checking with a satisfiability modulo theories (SMT) solver. Furthermore, we provide an implementation of RA and evaluate it by utilizing publicly available reference implementations of re-entrancy vulnerabilities [1], [2], [29]. Our results confirm that, unlike Oyente [23], RA can analyze precisely the vulnerability of deployed contracts to state-of-the-art re-entrancy attacks [29]. We have released the source codes of RA via GitHub (https://github.com/wanidon/RA) for reproducibility and as reference for future works.

Our main contribution is the creation of a new symbolic execution method named *symbolic re-entrancy emulation*, which emulates re-entrancy attacks by connecting different contracts with each other. As will be described in detail in Section 3, Oyente and its extensions [19], [33] do not support inter-contract analysis, for example, CFGs become fragments. On the other hand, we developed a module that localizes stored data on the blockchain in each execution path and a module that stacks a return address of an account information for each path. Using these modules, RA can support inter-contract analysis and emulate the behavior of re-entrancy attacks in a symbolic fashion via an internal implementation of a dummy contract, i.e., a contract that executes other contracts.

As another important contribution, we developed a new method named *vulnerability verification*, which verifies vulnerabilities by utilizing the Z3 SMT solver in the CFGs obtained from the symbolic re-entrancy emulation. In particular, on path conditions of the obtained CFGs, our method verifies whether program behavior on paths for executions with re-entrancy attacks is identical to that without the attacks, i.e., behavior on normal executions. Using the methods above, RA can completely eliminate false positives and false negatives (see Section 4 for details).

1.3 Paper Organization

The rest of this paper is organized as follows. A background of smart contracts and program analysis is presented in Section 2. Re-entrancy attacks and the technical difficulties in analyzing re-entrancy attacks are discussed in Section 3. The design of RA is presented in Section 4, and then its experimental evaluation is presented in Section 5. A review of related works is given in Section 6. Finally, a conclusion is presented in Section 7.

2. Technical Background

2.1 Ethereum Smart Contracts and EVM

In Ethereum, there are two kinds of accounts, namely, an externally owned account (EOA) and a contract account. EOAs have a private key that can be used to access the corresponding Ether or contracts. A contract account has smart contract code, which an EOA cannot have, and it does not have a private key. Instead, it is owned and controlled by the logic of its smart contract code. In Ethereum, a smart contract refers to an *immutable* computer program that is deployed on the blockchain and runs *deterministically* in the context of the EVM. The immutability property indicates that, similar to any data published on a general blockchain, smart contract codes can be considered as trustworthy, i.e., once deployed, they cannot be changed or deleted. The deterministic property indicates that the execution of the coded functions of smart contracts will produce the same result for anyone who runs them. Once deployed on the blockchain, a contract is self-enforcing and managed by the peers in the network, i.e., its functions are executed when the conditions in the contract are met. A smart contract is given an identity in terms of a contract address. Using this address, it can receive Ether and its functions can be executed. A contract is invoked when its contract address is the destination of a transaction, which is a signed message originating from an EOA, transmitted by the network, and recorded on the blockchain. Such a transaction causes a contract to run in the EVM using the transaction (and transaction's data) as input. The data indicate which specific function in the contract to run and what parameters to pass to that function. To incentivize peers to execute contract functions, Ethereum relies on *gas*, which is paid in Ether, to "fuel computations". The amount of gas needed to execute a transaction is relative to the complexity of the computations, thus also preventing infinite loops.

Smart contracts are typically written in a high-level language such as Solidity [2]. The source code is then compiled to low-level bytecode that runs in the EVM. The EVM is a simple stack-based architecture. Its instruction set is kept minimal to avoid incorrect implementations that could cause consensus problems. The EVM is a global singleton, i.e., it operates like a global, single-instance computer that runs in all peers in the network. Each peer runs a local copy of the EVM to validate the execution of contract functions, and the processed transactions and smart contracts are recorded on the blockchain.

2.2 Static Analysis of Programs

This paper focuses on the use of static analysis composed of CFGs, SMT solver, and symbolic execution. A CFG represents feasible paths of a program as a graph and is utilized for optimization by a compiler and static analysis of programs. Paths in a manner of sequential execution without branches are called *basic blocks* and are identical to nodes of a CFG. Likewise, paths which are feasible via branches or jumps are represented by edges to connect with nodes.

An SMT solver is a tool used for SMT problems. In contrast to satisfiability problems represented by proportional logic, SMT problems are represented by the first-order predicate logic which is more representative. By describing a specification to be verified in some logic formally, an SMT solver verifies whether a program satisfies the given specification.

Symbolic execution is a method that pseudo-executes a program by replacing information unspecified in the program itself with symbolic values to represent any value. Symbolic execution is composed of CFGs and an SMT solver, and it is suitable for analysis of smart contracts given that smart contracts utilize information on blockchains which are outside of program codes. Specifically, a condition to execute a path is called a *path condition*. Path conditions at the beginning of program execution are valid, and a restriction is newly added to the path conditions when a branch occurs. In a case where a path condition contains a symbolic value, executing either one or both paths according to a condition, i.e., the condition is satisfied or not, is decided by checking the satisfiability of the condition. Path conditions are often represented by first-order predicate logic, and their satisfiability is decided by an SMT solver described above.

3. Motivating Example and Technical Difficulties

In this section, we recall the fallback function and the re-entrancy attacks shown by Rodler et al. [29] as our motivating example and then discuss the technical difficulties in the analysis of the attacks.

3.1 Re-entrancy Problem

3.1.1 Fallback Function

Functions in Solidity are similar to classes in object-oriented languages. There are four types of Solidity functions, namely, `external`, `internal`, `public`, and `private`. A contract can have exactly one unnamed function called a *fallback function*, which cannot have any arguments, cannot return anything, and should have external visibility. The fallback function of a contract is executed whenever the contract receives Ether without any data included. To receive Ether and add it to the total balance of the contract, the fallback function must be marked `payable`. If the contract does not have a fallback function, then it cannot receive Ether through regular transactions and throws an exception. In other words, if a contract is intended to not receive Ether, then the payable in the fallback function can simply be removed. The fallback function is also triggered if someone tries to call a function that does not exist in the contract, and is often utilized for re-entrancy attacks.

3.1.2 Create-Based Re-Entrancy Attack

CREATE is an instruction that creates a new contract during execution of a contract (we call this the original contract for convenience). The new contract consists of initialization codes and the codes of its functions, and these codes are allocated by following a STOP instruction of the original contract. Once the initialization codes are executed, data is initialized and then bytecodes of the new contract are returned. These transitions via the initialization codes can be viewed as function calls by CREATE. Whenever a new contract is created, its constructor will be executed immediately. In the attacks by Rodler et al. [29], a newly generated contract by CREATE can issue further calls in its construc-

tor to other contracts, including malicious contracts, via CALL in the initialization codes. Here, the victim contract first creates a new contract and then updates its internal state. The newly created contract then calls a contract owned by an adversary. Consequently, the adversary maliciously withdraws from the victim contract via the re-entrancy.

3.1.3 Cross-Function Re-Entrancy Attack

Whenever function calls in which Ether is sent to an external contract account are executed, transactions are created by the CALL instruction. In doing so, some data are sent together with the transaction. The four most significant bytes correspond to a function ID of a caller function, which is obtained from a function name and a type name of variables. The callee contract obtains the function ID of the caller contract from the received transaction and then decides a function to be executed by checking if the ID is identical to that owned by the callee contract. If the callee contract does not own a function ID specified by a transaction, a fallback function is executed as described above. Cross-function re-entrancy attacks proposed by Rodler et al. [29] are launched over multiple functions of the victim contract. Specifically, cross-function re-entrancy attacks are launched by re-entering the same contract via a *different* function, whereas classical re-entrancy attacks are launched by re-entering the same contract via the same function.

3.2 Technical Difficulties

We present some technical issues that need to be solved for the analysis of the attacks described in the previous subsection. First, existing tools [19], [23], [33] do not provide support for generating running states of a callee contract via opcodes such as CREATE and CALL, which utilize an external contract. Consequently, in these tools, a caller contract is unidentified uniquely from the standpoint of a callee contract and vice versa. More concretely, in the reference implementation of re-entrancy attacks [29], an initialization code for the create-based re-entrancy attack is executed in the aforementioned manner. In doing so, the initialization code is on an *infeasible* path for existing tools. Moreover, because the main body of the created code is determined by a return value from the initialization code, the contract by CREATE is infeasible unless the initialization code is analyzed. Thus, the behavior of the created contract including CALL is unknown during offline analysis. Likewise, when an external contract is called by CALL for the cross-function re-entrancy attacks [29], analysis should be executed independently for each contract because running states of a callee contract are not generated. Consequently, utilizing path conditions for a caller contract is no longer meaningful to analyze the behavior of a callee contract. Furthermore, there are many candidates of function combinations, and thus analyzing vulnerability to cross-function re-entrancy attacks becomes difficult due to the potential state explosion [29].

Second, a method that evaluates vulnerabilities should be considered as well. Several analysis tools [7], [23], [26] report feasible paths but do not provide evaluation and detection of vulnerabilities. Accordingly, analysts often need to determine if a contract is vulnerable in a heuristic fashion, and hence many false positives and false negatives are produced. To eliminate false positives and false negatives, analysis tools should include a method that can evaluate vulnerabilities without requiring analysts to have prior knowledge of the attacks.

4. Design of RA

In this section, we present *Re-entrancy Analyzer* (*RA*), a new static analysis tool for re-entrancy attacks on Ethereum smart contracts. We first describe our design concept and then describe symbolic re-entrancy emulation and vulnerability verification as main processes, including their implementation.

4.1 Design Concept

The main idea of RA is to combine symbolic execution and an SMT solver in a *dual way*, i.e., *symbolic re-entrancy emulation* and *vulnerability verification*.

First, modules that *emulate* re-entrancy attacks based on symbolic execution are developed. Loosely speaking, by using an SMT solver to verify if conditions for calling to an external contract and generating basic blocks are satisfied, RA can identify which function is called. Consequently, RA can find path conditions via symbolic execution, which transits executions to each basic block recursively. Second, by utilizing the path conditions obtained from the emulation process above, RA *verifies* if a resultant running state of a function call based on a fallback function is equivalent to the original behavior, i.e., without the fallback function, on the CFG obtained from the emulation. The verification is done by the SMT solver. While symbolic executions in literature [19], [23], [33] report only program behavior via CFGs obtained within a *single* contract, RA emulates re-entrancy attacks including *inter-contract* behavior and then it verifies the vulnerability of these contracts to re-entrancy attacks. Consequently, in contrast to the early literature, while the coverage of the analysis for re-entrancy attacks has improved drastically, RA can analyze the vulnerabilities precisely, i.e., without false positives and false negatives and without requiring analysts to have prior knowledge of attack patterns, including state-of-the-art re-entrancy attacks [29].

4.2 Tool Overview

The overview of RA is as follows. First, for the symbolic re-entrancy emulation, RA generates CFGs from bytecodes whereby a newly created/called contract is represented within a flow of the main contract via symbolic execution. Then, for vulnerability verification, RA verifies whether the bytecodes are vulnerable to re-entrancy attacks by utilizing the Z3 SMT solver in accordance with path conditions obtained from the symbolic re-entrancy emulation. To do this, the following notions are defined for RA:

- *Local-world state* is owned by a basic block in local and stores global information, such as balance or storage.
- *Call stack* is a stack that stores a return address to a basic block.
- *Contract queue* is a queue that stores bytecodes of a contract to be analyzed.

An overview of RA including the notions is presented in **Fig. 1**. RA mainly consists of three modules, namely, *CFManager*, *VM*, and *Verifier*. The symbolic re-entrancy emulation process is
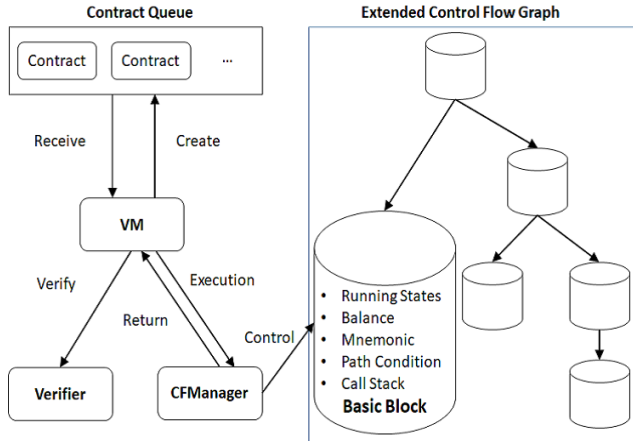
Fig. 1   Overview of RA.

mainly conducted by CFManager and VM. On the other hand, the vulnerability verification process is executed by Verifier assisted by VM. We describe the role of each module below.

CFManager handles feasible paths by directly operating data structures in a basic block for a CFG. A basic block contains the local-world state, the call stack, as well as mnemonic of instructions, running states, and path conditions. These give us information retrieved from the blockchain and return values, i.e., information obtained as a result of each caller/callee contract. Consequently, symbolic execution of each block can be covered even through an external contract. First, VM receives a contract to be analyzed from the contract queue, and then CFManager executes instructions on basic blocks. Finally, Verifier verifies existence of re-entrancy vulnerability by utilizing the Z3 SMT solver with the information obtained by VM. Based on these modules, the symbolic emulation for inter-contract analysis is provided and then the verification of vulnerabilities is executed. To use RA and analyze contracts, analysts only need to provide EVM bytecodes of target contracts as input and they do not need to have prior knowledge about re-entrancy attacks. The details of each process are presented below.

4.3   Symbolic Re-entrancy Emulation

The goal of this process is to generate a CFG completely via emulation of re-entrancy attacks. In particular, the CFManager generates a CFG to represent inter-contract behavior by recording basic blocks with transitions as edges. We call such a CFG an *extended CFG (ECFG)* for convenience. In ECFGs, CREATE and CALL instructions are utilized as separators of basic blocks in addition to JUMP and STOP instructions. For convenience, let contracts to be executed by CREATE and CALL be callers, and let contracts to be created or called be callees. When CREATE appears, the symbolic execution is transited to its callee contract, i.e., the initialization code. Similarly, when CALL appears, the execution is transited to a function of its callee contract. When STOP appears within the callee contracts described above, the execution is returned to the caller contract. These transitions of contracts are managed by VM.

We now describe the emulation process by RA in detail. For instance, for analysis of the create-based re-entrancy attacks, VM extracts an initialization code of a contract from variables of
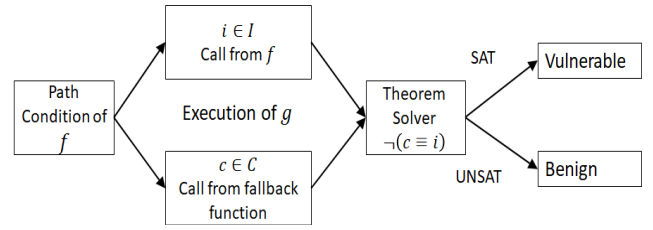

Fig. 2   Process of vulnerability verification.

CREATE by specifying basic blocks via the Z3 SMT solver, and CFManager transits the execution to those blocks in accordance with branches decided by the Z3 SMT solver. By obtaining each block and connecting them, CALL in the initialization code can be identified. Moreover, VM can register the callee contract obtained by the initialization code in the contract queue, and thus the whole contract can be symbolically emulated in a recursive manner. On the other hand, for analysis of the cross-function re-entrancy attacks, a function ID is symbolically executed as a symbolic value, and then Verifier extracts the function ID to be executed by the Z3 SMT solver. By giving the ID at the beginning execution and re-execution, combination of any functions is representative.

4.4   Vulnerability Verification

The vulnerability verification is done by Verifier with path conditions obtained from the emulation described in the previous subsection. Let functions to be verified be f, g, where f contains a function call during the execution. We also denote by I a set of path conditions where g is executed by taking over a result in the execution of f, and by C a set of path conditions where f calls a fallback function in a manner that the fallback function calls g and then f is executed again with the result of g. The Verifier module can receive I and C from the VM module. Then, the Z3 SMT solver verifies whether a program is vulnerable or not as follows:

$$\exists c \in C, \forall i \in I : \neg(c \equiv i). \tag{1}$$

Using Eq. (1), the Z3 solver finds $c \in C$ such that c is not equal to any $i \in I$, i.e., the program is vulnerable if $\neg(c \equiv i)$, and benign otherwise.

The procedure of the vulnerability verification is shown in **Fig. 2**. The withdraw process is not executed when g is executed after executing f in a normal way, i.e., executions in I. On the other hand, the withdraw process is executed due to the re-entrancy on f whereby a fallback function calls g, i.e., executions in C. RA decides that a contract is vulnerable if program behaviors are equivalent in these executions. In particular, RA checks if there is no case where the behaviors are equivalent. Intuitively, the main difference between these cases is whether instructions following CALL in f are executed before g. Specifically, g contains a branch that determines whether the withdraw process is executed in accordance with changing states on the blockchain, where path conditions at the end of executions for both cases are different from each other.

4.5 Implementation

The main techniques for implementing the symbolic re-entrancy emulation of RA are presented as the technical parts of the implementation below. In particular, we describe how basic blocks are controlled and function IDs are extracted to reduce potential state explosion, which is one of the problems for static analysis tools according to Rodler et al. [29]. Several techniques for improving the performance are presented as well. We released the source codes of the whole implementation publicly via GitHub (https://github.com/wanidon/RA).

4.5.1 Environment

RA is implemented on Python 3.7 and three modules, i.e., z3, pysha3, and Graphviz. The z3 module provides APIs for the use of the Z3 SMT solver on Python. The pysha3 module enables the use of the Keccak256 hash function as a SHA-3 module. Finally, Graphviz provides support for drawing CFGs. We also note that RA is independent of any specific compiler because it uses bytecodes as input.

4.5.2 Control of Basic Blocks for CFManager

The method that represents a branch with a conditional jump operation JUMPI is as follows. First, RA adopts the depth-first search for finding feasible paths, and the CFManager owns a data structure named *dfs_stack* which stores candidates of basic blocks to be searched. Then, the CFManager pops two variables for JUMPI from the dfs_stack, i.e., a jump address and a condition c for the jump where c is determined by true or false. When c is a symbolic value, the z3 module decides whether $\neg c$ is satisfiable. If so, there exists a path that does not contain jump. Then, a new block is generated by copying the current basic block and incrementing a program counter, and the new block is newly recognized as the basic block. Next, the solver decides whether c is satisfiable. If so, there exists a path that contains jump. Then, the current basic block is copied as a jump block and the program counter is set as the jump address. In the case that there exists a path with jump but a path without jump does not exist, the current basic block is set as a jump block. In contrast, if both a path with jump and that without jump exist, then the jump block itself is stored in the dfs_stack.

4.5.3 Extraction of Function ID for Verifier

Decision of a function ID for Verifier is implemented as follows. First, a symbolic value $function\_id$ is assigned with the four most significant bytes of any transaction. The value is loaded onto a contract to be analyzed, and hence paths are branched for each function by comparing them with actual function IDs for the contract. Then, the function ID can be obtained because a solution satisfying $function\_id$ is obtained by deciding the satisfiability of path conditions for all paths with the z3 module. If a CALL instruction appears during the execution, CALLABLE is recorded as a state of a basic block. The state of the basic block is inherited by all the descendant paths. In extracting an actual function ID, ID of a function with function call is recorded if the end state of a basic block is CALLABLE.

4.5.4 Speed-up Techniques

The entire performance of RA was improved by using four optimization techniques. First, RA can verify combinations of function calls described in Eq. (1) in parallel. Second, a constraint is given on push to a stack. A CALL instruction normally requires push to a stack in accordance with success of a function call, i.e., 0 or 1. In contrast, a re-entrancy attack is always executed after the success of a function call. Thus, RA always pushes 1, i.e., the success of the function call, to a stack as assumed in the success of CALL instruction. This enables RA to reduce the number of path conditions as a speed-up technique. Third, because CALL is required to always succeed, a constraint, i.e., a balance for any contract account is a positive value, is given on path conditions only at the end states. The computational complexity can be reduced drastically by giving the constraint at the end states instead of at each change of the balance. Surprisingly, for analysis of several contracts, the computational performance becomes ten times faster by using this constraint although we omit the details. Finally, RA stops to analyze paths for some stop instruction without the rollback process. In particular, a stop instruction called REVERT contains the rollback about the blockchain related to a given transaction. However, to the best of our knowledge, re-entrancy attacks never occur in this case, thus RA does not analyze the remaining parts of such paths.

5. Evaluation and Discussion

In this section, we will show that RA can analyze inter-contract control flows precisely in comparison with execution of Oyente [23]. Then, we will show the computational performance of RA for analysis. Finally, we will provide a discussion on the comparison of RA with other tools and the current limitations of RA.

5.1 Case Studies

As case studies, we test RA with reference implementations of the re-entrancy attacks by Rodler et al. [29] and known re-entrancy vulnerabilities [1], [2]. The goal of these studies is to clarify the ability of RA to perform inter-contract analysis. The codes of the reference implementations of the re-entrancy attacks [29] are publicly available [*2].

5.1.1 Creation of Contracts

Consider the output of RA shown in Fig. A·1 [*3]. On a CFG output by RA for a contract created via CREATE, the execution is transited to a callee contract on the basic block (denoted by the red box), and the execution is returned to the basic block (denoted by the green box). In contrast, Oyente cannot transit execution into the initialization code, i.e., a created contract, in a symbolic manner. The remaining parts of the CFG are identical to a CFG output from Oyente. Consequently, RA can handle a function call by extracting the initialization codes on CREATE instructions.

Consider the output of RA shown in Fig. A·2 [*4]. On a CFG output by RA on analysis of the reference implementation of the create-based re-entrancy attack, the execution is transited/returned to/from a callee contract on the basic block. The figure is an output of RA for the Fund contract [2], and the same figure is obtained even when the Fund function is called by the CREATE instruction. This confirms that, unlike Oyente, RA

---

*2   https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns
*3   This figure is shown in the Appendix.
*4   This figure is shown in the Appendix.

Table 1   Results of Analysis of Contracts Vulnerable to Re-Entry Attacks: The Benign Functions and Vulnerable Functions columns represent the number of benign functions and vulnerable functions in the contract, respectively. The true positive rate is denoted by TPR, false positive rate by FPR, true negative rate by TNR, and false negative rate by FNR.

Contract	Code Length [Byte]	Benign Functions	Vulnerable Functions	TPR [%]	FPR [%]	TNR [%]	FNR [%]
Fund [2]	356	0	1	100	0	0	0
KnownReentrancy [1]	365	0	1	100	0	0	0
Bank [29]	1,694	2	1	100	0	100	0
Token [29]	2,516	11	1	100	0	0	0
KnownCrossFunction [1]	680	1	1	100	0	100	0

provides analysis of create-based re-entrancy attacks in an inter-contract fashion.

5.1.2   Call of Contracts

We omit the output of RA in this case due to its length and describe only the intuition. Suppose that an adversary utilizes a contract with only CALL instructions. In particular, because there are two basic blocks, two function calls and two transitions to the caller are identified according to the analysis results of RA. These results confirm that RA provides analysis of cross-function re-entrancy attacks in an inter-contract fashion.

5.2   Vulnerability Analysis

The performance of RA is evaluated by using it to analyze the vulnerability of a number of deployed contracts to re-entrancy attacks. In particular, the target contracts included in the evaluation are the *Fund* contract in the official document of Solidity [2], a contract named *KnownReentrancy* obtained from the code of "Reentrancy on a Single Function" published on the webpage of "Ethereum Smart Contract Best Practices" [1], the *Bank* contract based on create-based re-entrancy attack [29], the *Token* contract based on cross-function re-entrancy attack [29], and a contract named *KnownCrossFunction* obtained from the code of "Cross-function Reentrancy" published on "Ethereum Smart Contract Best Practices". Both Fund and KnownReentrancy contain a single contract. Bank contains three contracts, two of which are benign, i.e., not vulnerable. Token contains twelve contracts, eleven of which are benign and only one contract is vulnerable. Finally, KnownCrossFunction contains one vulnerable contract and one benign contract because the cross-function re-entrancy attack needs "cross" calls between different contracts. Several contracts include multiple functions and hence analysis of re-entrancy attacks becomes complicated due to combinations of function calls as described in Section 3.2. In doing so, the goal of this evaluation is to determine whether RA can precisely identify both vulnerable contracts and benign contracts.

The results are shown in **Table 1**. According to the table, RA can precisely verify all the existing vulnerabilities without false positives and false negatives. In other words, in addition to being able to analyze contracts that are vulnerable to re-entrancy attacks, RA can also analyze contracts that are not vulnerable, as confirmed by the true negative rate. This precise evaluation for verification of vulnerabilities was obtained by the use of the Z3 SMT solver with path conditions obtained from symbolic execution.

Table 2   Computational Time of RA for Analysis: The Combinations of Functions column represents the number of combinations of function calls in the contract.

Contract	Code Length [Byte]	Combinations of Functions	Time [sec]
Fund [2]	356	1	20.750
Known Reentrancy [1]	365	1	23.003
Bank [29]	1,694	3	47.579
Token [29]	2,516	12	519.676
KnownCross Function [1]	680	2	37.705

Table 3   Comparison of RA and Other Analysis Tools: ● indicates the tool can detect the attack, ○ indicates the tool cannot detect the attack, and ⊘ indicates the tool can potentially detect the attack but did not provide a discussion in its paper.

Tool	Static Analysis	Cross-Function	Create-Based	Analysis Target
Oyente [23]	Static	○	○	EVM
Securify [33]	Static	○	○	EVM
Annotary [35]	Static	⊘	⊘	Solidity
Sereum [29]	Dynamic	●	●	EVM
ÆGIS [12]	Dyanmic	●	⊘	EVM
RA	Static	●	●	EVM

5.3   Computational Performance

We now present the computational performance of RA for analysis of re-entrancy attacks. The performance was measured by utilizing the time.perf_counter function as an average of ten executions. The environment for measurement is as follows: iMac 21.5-inch, 2017 with 3.6 GHz Intel Core i7 processor, 32 GB memory, and Radeon Pro 560 4 GB as GPU. The measurement was done by implementing parallel-processing. As can be seen in **Table 2**, the computational time for analysis becomes longer in proportion to the number of function calls. Surprisingly, analysis of Bank is fast even with its code length because Bank is based on create-based re-entrancy attack, i.e., the majority of its codes to create a contract and the function call are restricted to call the created contract. In contrast, Token is based on cross-function re-entrancy attack and its code length and variation of function calls are large. Nevertheless, RA was still able to analyze Token within reasonable time.

5.4   Comparison to Other Analysis Tools

Table 3 shows a comparison of different tools in terms of their ability to detect re-entrancy attacks discovered by Rodler et al. [29]. Oyente [23] and Securify [33] cannot detect such re-entrancy attacks. In particular, Oyente fails to detect the vulner-

abilities, while Securify produces false alerts due to its conservative policy. Sereum and ÆGIS can potentially detect the vulnerabilities, but they are based on dynamic analysis. Finally, Annotary is a static analysis tool that can deal with CREATE and CALL instructions. Thus, it can potentially detect the re-entrancy attacks, but its paper did not discuss re-entrancy attacks. Moreover, Annotary's analysis target are Solidity codes and thus it may experience difficulties in analyzing deployed contracts.

5.5 Computational Complexity for Vulnerability Verification

Let the number of functions with function call be F_n, the number of those paths be F_p, the number of any function be G_n, and the number of those paths be G_p. Here, the number of functions to be executed in the first step is F_n and its resulting number of end states for each execution is F_p. Then, in the second step, G_n functions are executed in proportion to the number of end states, and the number of the end states of G_n is G_p. Consequently, the order complexity to obtain path conditions at the end of the process is $O(F_n F_p G_n G_p)$. By parallelizing the process, the complexity becomes $O(G_n G_p)$ because all combinations of function calls can be parallelized. To compute path conditions using Eq. (1), RA decides whether two sets, i.e., C and I, of the path conditions obtained from combinations of one function have an identity relation between $i \in I$ and $c \in C$. In doing so, the order complexity is $O(F_n F_p G_n^2 G_p^2)$, which is smaller than the complexity to obtain path conditions at the end states. The execution time for analysis becomes longer in proportion to the number of contracts to be analyzed because the order complexity described above is polynomial time. In contrast, RA can be expected to be utilized for analysis of contracts developed by a user in realistic time.

5.6 Extension to Analysis of Delegated Re-Entrancy Attacks

Rodler et al. [29] presented delegated re-entrancy attacks where a contract invokes another contract as a library within instructions that utilize contracts as an external library, e.g., DELEGATECALL or CALLCODE. These instructions are currently not implemented in RA, but RA can be extended to analyze delegated re-entrancy attacks by introducing these instructions. The main technical difficulty in analysis of delegated re-entrancy attacks is that the library contract that will be used is unknown [29]. This difficulty is also a problem in analyses of create-based and cross-function re-entrancy attacks, which have been overcome already in RA.

5.7 Limitations

The current implementation of RA has three limitations. First, analysis of gas is not considered. Thus, contracts that restrict gas consumption may not be analyzed precisely. Second, a case where multiple contracts are tightly coupled with each other, i.e., more than two contracts are strongly interdependent, is out of the scope of RA. Specifically, instructions that utilize other contracts as an external library are not implemented. Consequently, the number of contracts to be analyzed in a single execution should be at most one, and the number of callee contracts is limited to a single contract. An improved implementation of RA, which is our

ongoing work, may be able to overcome the limitation described above. Finally, bytecodes of contracts cannot include symbolic values. Accordingly, a case where created contracts are different for each execution is not considered. Improving the points described above is our ongoing work.

6. Related Works

In this section, we recall early literature on security analysis of Ethereum smart contracts in terms of symbolic execution and formal methods as static analysis. Then, we describe several multidisciplinary approaches proposed in the past few years as additional related works. Interested readers are advised to read the survey papers [11], [37] for details on EVM analysis.

6.1 Symbolic Execution

Symbolic execution of Ethereum smart contracts was originally started by Oyente [23]. Although there are many subsequent works [8], [21], [26], [31], these works do not perform inter-contract analysis. Furthermore, analysis of re-entrancy attacks is often heuristic and thus Oyente often produces many false positives. Extensions of Oyente that support readability of outputs from symbolic execution have been proposed [24], [25], and the usability of RA can be potentially improved in a similar way.

The closest work to RA is Annotary [35], which can analyze inter-contract behavior via both symbolic executions of EVM bytecodes and the Z3 SMT solver. The major difference of Annotary from RA is that Annotary mainly targets analysis of Solidity codes. In other words, RA mainly checks if deployed contracts are secure or not, while Annotary supports developers in implementing secure codes. Compared to RA, Annotary has several advantages. In particular, according to Weiss et al. [35], Annotary's focus on the analysis of Solidity code brings the following two advantages. First, an analyst can use annotations to express invariants and restrictions directly in the Solidity source code and then use these annotations to find a feasible path on symbolic execution effectively. On the other hand, RA focuses on analysis of EVM bytecodes and hence cannot handle Solidity source code, i.e., RA cannot analyze Solidity source codes because it targets EVM bytecodes. Therefore, Annotary has an advantage in finding a feasible path effectively by its use of annotations. Second, Solidity source codes can be operated and written using a typical editor, and then a software developer can incorporate a plugin such that an analysis tool coordinates with existing editors. For instance, Annotary has provided a Sublime Text plugin. It means that, for example, a developer can analyze the Solidity source codes written by him-/herself. The analysis of EVM bytecodes requires a developer to compile the source codes to get the EVM bytecodes or to obtain the EVM bytecodes of deployed contracts, and therefore it would be challenging to embed RA into a typical editor. Providing a plugin is thus another advantage provided by Annotary. Although the authors of Annotary did not consider the new attacks by Rodler et al. [29], we consider their idea and work to be elegant nonetheless.

ETHBMC [13] and VerX [27] are recent state-of-the-art works that verify properties of Ethereum. VerX is similar to Annotary in terms of taking Solidity codes as input and dealing with exter-

nal contracts. Similar to Annotary, the new attacks by Rodler et al. [29] were not considered in VerX. On the other hand, ETHBMC takes EVM bytecodes for symbolic executions and its motivation is rather close to that of RA. However, ETHBMC mainly focuses on parity vulnerability.

6.2 Formal Methods

Formal verification of EVM was motivated by Bhargavan et al. [6], and EVM was correctly formalized as KEVM [18]. However, verification of the security is challenging in general, and the existing works [3], [4], [19], [33] do not provide support for inter-contract analysis, which is the main target of our work. As more theoretical approach, Grishchenko et al. [15], [16] formalized several attacks, including re-entrancy attacks, as well as formalization of EVM. Their formalization inspired the vulnerability verification of RA. As the latest work, Grishchenko et al. presented a formal verification tool [30] that can deal with inter-contract analysis. This work is concurrent with ours.

6.3 Multidisciplinary Approaches

NeuCheck [22] can rapidly analyze Solidity source codes by extracting a syntax tree on a cross-platform environment. TokenScope [9] can detect vulnerabilities by identifying tokens that have a different specification from the ERC20 token. SMARTSHIELD [36] is a bytecode rectification system that fixes security-related bugs automatically. SmartEmbed [14] and VulDeeSmartContract [28] are automated vulnerability detection frameworks based on machine learning for the natural language processing. TEETHER [20] is a penetration tool for exploit generation of Ethereum smart contracts. The techniques in these multidisciplinary works can be used to potentially improve RA. Finally, ILF [17] and ContractWard [34] use a combination of machine learning and symbolic execution to improve testing coverage. These works are expected to have improved performance if they deploy RA as a building block.

7. Conclusion

In this paper, we introduced RA, a static analysis tool that provides inter-contract analysis of the EVM bytecodes of Ethereum smart contracts to detect vulnerabilities to state-of-the-art re-entrancy attacks [29]. Using RA, analysts do not need to have prior knowledge of re-entrancy attacks to detect them. To create RA, we designed modules that represent inter-contract CFGs by the symbolic re-entrancy emulation and the vulnerability verification with the Z3 SMT solver to verify the re-entrancy vulnerability of deployed contracts. We also conducted experiments on deployed contracts and confirmed the performance of RA by precisely identifying combinations of contracts with and without vulnerabilities. The aforementioned performance could be obtained by the high-level combination of the symbolic execution and the Z3 SMT solver.

As future work, we plan to extend RA for a case in which a fraction of contracts to be verified and external contracts become many-to-many. If successful, RA will be able to deal with more complicated attacks that can be proposed in the future. Moreover, we plan to design capabilities that check for vulnerabilities aside from re-entrancy, such as time-dependent vulnerability which utilizes a timestamp for each block. We believe that the inter-contract analysis capability of RA can potentially identify such a vulnerability. Finally, we will also try to improve the computational performance of RA by reusing parts of the computations for vulnerability verification.

References

[1] Known Attacks, available from ⟨https://consensys.github.io/smart-contract-best-practices/known_attacks/⟩.

[2] Security Considerations Solidity 0.5.11 documentation, available from ⟨https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html⟩.

[3] Albert, E., Gordillo, P., Livshits, B., Rubio, A. and Sergey, I.: EthIR: A Framework for High-Level Analysis of Ethereum Bytecode, *Proc. ATVA 2018*, Lahiri, S.K. and Wang, C. (Eds.), LNCS, Vol.11138, pp.513–520, Springer (2018).

[4] Amani, S., Bégel, M., Bortin, M. and Staples, M.: Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL, *Proc. CPP 2018*, pp.66–77, ACM (online), DOI: 10.1145/3167084 (2018).

[5] Atzei, N., Bartoletti, M. and Cimoli, T.: A survey of attacks on ethereum smart contracts (sok), *Proc. POST 2017*, LNCS, Vol.10204, pp.164–186, Springer (2017).

[6] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al.: Formal verification of smart contracts: Short paper, *Proc. PLAS 2016*, pp.91–96, ACM (2016).

[7] Chang, J., Gao, B., Xiao, H., Sun, J., Cai, Y. and Yang, Z.: sCompile: Critical path identification and analysis for smart contracts, *Proc. ICFEM*, LNCS, Vol.11852, pp.286–304, Springer (2019).

[8] Chen, T., Li, X., Luo, X. and Zhang, X.: Under-optimized smart contracts devour your money, *Proc. SANER 2017*, pp.442–446, IEEE (2017).

[9] Chen, T., Zhang, Y., Li, Z., Luo, X., Wang, T., Cao, R., Xiao, X. and Zhang, X.: TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum, *Proc. CCS 2019*, pp.1503–1520, ACM (2019).

[10] Chinen, Y., Yanai, N., Cruz, J.P. and Okamura, S.: RA: Hunting for Re-Entrancy Attacks in Ethereum Smart Contracts via Static Analysis, *Proc. IEEE Blockchain 2020*, pp.327–336, IEEE (2020).

[11] Di Angelo, M. and Salzer, G.: A survey of tools for analyzing ethereum smart contracts, *Proc. DAPPCON 2019*, pp.69–78, IEEE (2019).

[12] Ferreira Torres, C., Steichen, M., Norvill, R., Fiz Pontiveros, B. and Jonker, H.: ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks, *Proc. AsiaCCS 2020*, pp.584–597, ACM (2020).

[13] Frank, J., Aschermann, C. and Holz, T.: ETHBMC: A Bounded Model Checker for Smart Contracts, *Proc. Usenix Security 2020*, USENIX Association (2020) (online), available from ⟨https://www.usenix.org/conference/usenixsecurity20/presentation/frank⟩.

[14] Gao, Z., Jiang, L., Xia, X., Lo, D. and Grundy, J.: Checking Smart Contracts with Structural Code Embedding, *IEEE Trans. Software Engineering*, p.1 (2020).

[15] Grishchenko, I., Maffei, M. and Schneidewind, C.: Foundations and Tools for the Static Analysis of Ethereum Smart Contracts, *Proc. CAV 2018*, LNCS, Vol.10981, pp.51–78, Springer (2018).

[16] Grishchenko, I., Maffei, M. and Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts, *Proc. POST 2018*, LNCS, Vol.10804, pp.243–269, Springer (2018).

[17] He, J., Balunoviundefined, M., Ambroladze, N., Tsankov, P. and Vechev, M.: Learning to Fuzz from Symbolic Execution with Application to Smart Contracts, *Proc. CCS 2019*, pp.531–548, ACM (2019).

[18] Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., et al.: KEVM: A complete formal semantics of the ethereum virtual machine, *Proc. CSF 2018*, pp.204–217, IEEE (2018).

[19] Kalra, S., Goel, S., Dhawan, M. and Sharma, S.: ZEUS: Analyzing Safety of Smart Contracts, *Proc. NDSS 2018*, Internet Society (2018).

[20] Krupp, J. and Rossow, C.: teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts, *Proc. USENIX Security 2018*, pp.1317–1333, USENIX Association (2018).

[21] Liu, H., Liu, C., Zhao, W., Jiang, Y. and Sun, J.: S-gram: Towards semantic-aware security auditing for ethereum smart contracts, *Proc. ASE 2018*, pp.814–819, ACM (2018).

[22] Lu, N., Wang, B., Zhang, Y., Shi, W. and Esposito, C.: NeuCheck: A more practical Ethereum smart contract security analysis tool, *Software: Practice and Experience*, Vol.2019, pp.1–20 (2019) (online), available from ⟨https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2745⟩.

[23] Luu, L., Chu, D.-H., Olickel, H., Saxena, P. and Hobor, A.: Making smart contracts smarter, *Proc. CCS 2016*, pp.254–269, ACM (2016).

[24] Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T. and Dinaburg, A.: Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts, *Proc. ASE 2019*, pp.1186–1189, IEEE (2019).

[25] Mueller, B.: Smashing smart contracts, *9th HITB Security Conference* (2018).

[26] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P. and Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale, *Proc. ACSAC 2018*, pp.653–663, ACM (2018).

[27] Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D. and Vechev, M.: Verx: Safety verification of smart contracts, *Proc. IEEE S&P 2020*, pp.414–430, IEEE (2020).

[28] Qian, P., Liu, Z., He, Q., Zimmermann, R. and Wang, X.: Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models, *IEEE Access*, Vol.8, pp.19685–19695 (2020).

[29] Rodler, M., Li, W., Karame, G.O. and Davi, L.: Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks, *Proc. NDSS 2019*, Internet Society (2019).

[30] Schneidewind, C., Grishchenko, I., Scherer, M. and Maffei, M.: EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts, *Proc. CCS 2020*, pp.621–640, ACM (2020).

[31] Torres, C.F., Schütte, J., et al.: Osiris: Hunting for integer bugs in ethereum smart contracts, *Proc. ACSAC 2018*, pp.664–676, ACM (2018).

[32] Torres, C.F. and Steichen, M.: The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts, *Proc. Usenix Security 2019*, pp.1591–1607, Usenix Association (2019).

[33] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F. and Vechev, M.: Securify: Practical security analysis of smart contracts, *Proc. CCS 2018*, pp.67–82, ACM (2018).

[34] Wang, W., Song, J., Xu, G., Li, Y., Wang, H. and Su, C.: ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts, *IEEE Trans. Network Science and Engineering*, p.1 (Early Access) (2020).

[35] Weiss, K. and Schütte, J.: Annotary: A Concolic Execution System for Developing Secure Smart Contracts, *Proc. ESORICS 2019*, LNCS, Vol.11735, pp.747–766, Springer (2019).

[36] Zhang, Y., Ma, S., Li, J., Li, K., Nepal, S. and Gu, D.: SMARTSHIELD: Automatic Smart Contract Protection Made Easy, *Proc. SANER 2020*, pp.23–34, IEEE (2020).

[37] Zou, W., Lo, D., Kochhar, P.S., Le, X.-B. D., Xia, X., Feng, Y., Chen, Z. and Xu, B.: Smart Contract Development: Challenges and Opportunities, *IEEE Transactions on Software Engineering*, p.1 (2019).

Appendix

A.1   Outputs of RA

Figure A·1 shows CFGs output by RA on analysis of contracts with CREATE instruction. **Figure A·2** shows CFGs output by RA for the reference implementation of create-based re-entrancy attacks.

Editor's Recommendation

This paper proposes a static analysis tool named RA (Re-entrancy Analyzer), a combination of symbolic execution and equivalence checking to analyze vulnerability of smart contracts to re-entrancy attacks. This research is necessary and important for smart contracts. In particular, this paper shows the experiment results on deployed contracts and includes a sufficient amount of survey. The paper gives insights to readers in this research field and thus is selected as a recommended paper.
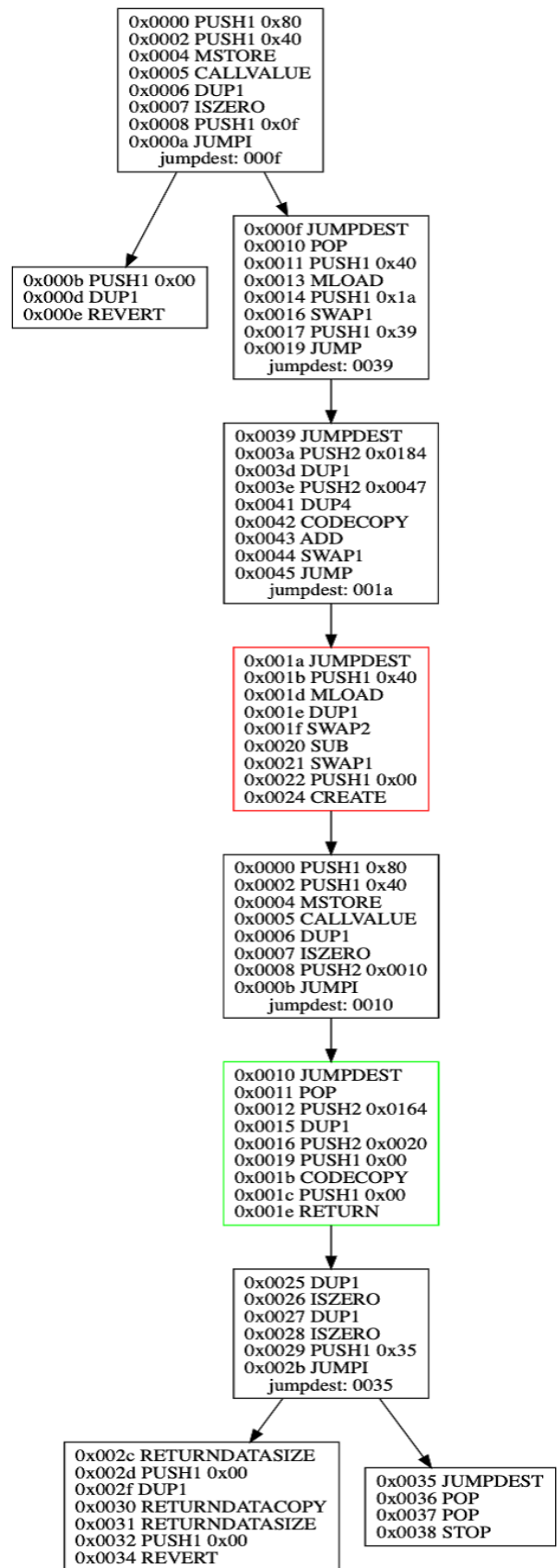
(Chief examiner of SIGSCEC Toshihiro Yamauchi)

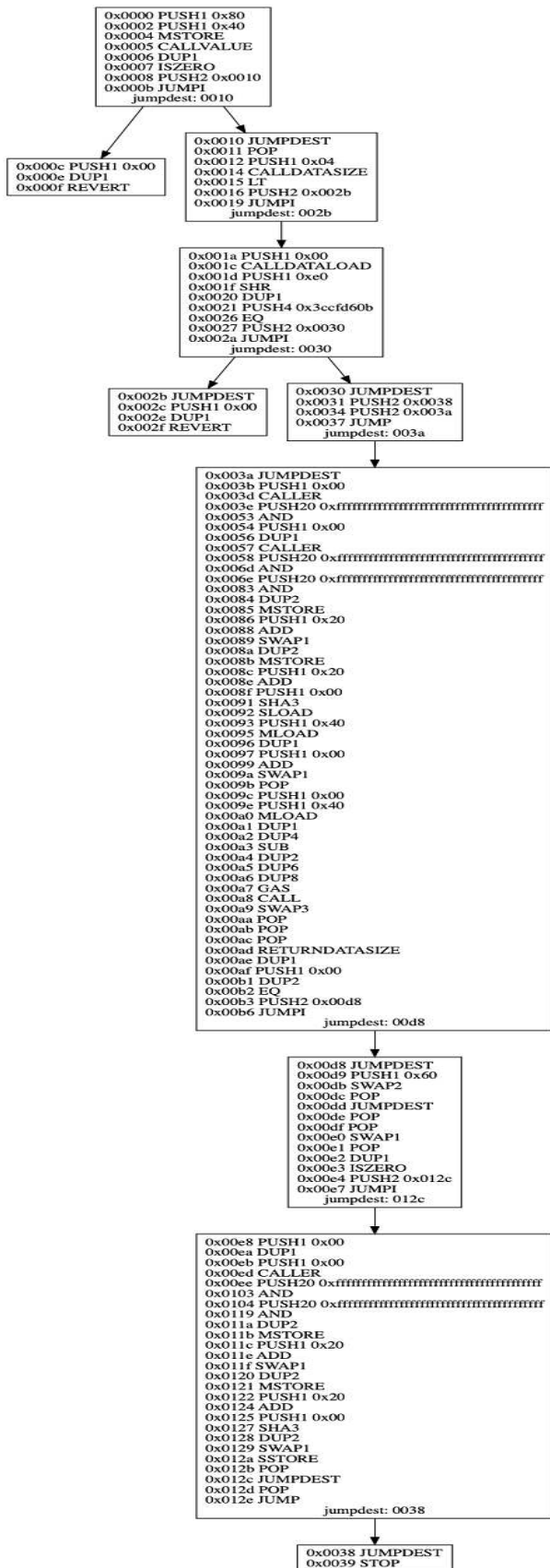**Fig. A·1**   Control flow graph obtained from CREATE instructions.

```
0x0000 PUSH1 0x80
0x0002 PUSH1 0x40
0x0004 MSTORE
0x0005 CALLVALUE
0x0006 DUP1
0x0007 ISZERO
0x0008 PUSH2 0x0010
0x000b JUMPI
        jumpdest: 0010

0x000c PUSH1 0x00
0x000e DUP1
0x000f REVERT

0x0010 JUMPDEST
0x0011 POP
0x0012 PUSH1 0x04
0x0014 CALLDATASIZE
0x0015 LT
0x0016 PUSH2 0x002b
0x0019 JUMPI
        jumpdest: 002b

0x001a PUSH1 0x00
0x001c CALLDATALOAD
0x001d PUSH1 0xe0
0x001f SHR
0x0020 DUP1
0x0021 PUSH4 0x3ccfd60b
0x0026 EQ
0x0027 PUSH2 0x0030
0x002a JUMPI
        jumpdest: 0030

0x002b JUMPDEST
0x002c PUSH1 0x00
0x002e DUP1
0x002f REVERT

0x0030 JUMPDEST
0x0031 PUSH2 0x0038
0x0034 PUSH2 0x003a
0x0037 JUMP
        jumpdest: 003a

0x003a JUMPDEST
0x003b PUSH1 0x00
0x003d CALLER
0x003e PUSH20 0xffffffffffffffffffffffffffffffffffffffff
0x0053 AND
0x0054 PUSH1 0x00
0x0056 DUP1
0x0057 CALLER
0x0058 PUSH20 0xffffffffffffffffffffffffffffffffffffffff
0x006d AND
0x006e PUSH20 0xffffffffffffffffffffffffffffffffffffffff
0x0083 AND
0x0084 DUP2
0x0085 MSTORE
0x0086 PUSH1 0x20
0x0088 ADD
0x0089 SWAP1
0x008a DUP2
0x008b MSTORE
0x008c PUSH1 0x20
0x008e ADD
0x008f PUSH1 0x00
0x0091 SHA3
0x0092 SLOAD
0x0093 PUSH1 0x40
0x0095 MLOAD
0x0096 DUP1
0x0097 PUSH1 0x00
0x0099 ADD
0x009a SWAP1
0x009b POP
0x009c PUSH1 0x00
0x009e PUSH1 0x40
0x00a0 MLOAD
0x00a1 DUP1
0x00a2 DUP4
0x00a3 SUB
0x00a4 DUP2
0x00a5 DUP6
0x00a6 DUP8
0x00a7 GAS
0x00a8 CALL
0x00a9 SWAP3
0x00aa POP
0x00ab POP
0x00ac POP
0x00ad RETURNDATASIZE
0x00ae DUP1
0x00af PUSH1 0x00
0x00b1 DUP2
0x00b2 EQ
0x00b3 PUSH2 0x00d8
0x00b6 JUMPI
        jumpdest: 00d8

0x00d8 JUMPDEST
0x00d9 PUSH1 0x60
0x00db SWAP2
0x00dc POP
0x00dd JUMPDEST
0x00de POP
0x00df POP
0x00e0 SWAP1
0x00e1 POP
0x00e2 DUP1
0x00e3 ISZERO
0x00e4 PUSH2 0x012c
0x00e7 JUMPI
        jumpdest: 012c

0x00e8 PUSH1 0x00
0x00ea DUP1
0x00eb PUSH1 0x00
0x00ed CALLER
0x00ee PUSH20 0xffffffffffffffffffffffffffffffffffffffff
0x0103 AND
0x0104 PUSH20 0xffffffffffffffffffffffffffffffffffffffff
0x0119 AND
0x011a DUP2
0x011b MSTORE
0x011c PUSH1 0x20
0x011e ADD
0x011f SWAP1
0x0120 DUP2
0x0121 MSTORE
0x0122 PUSH1 0x20
0x0124 ADD
0x0125 PUSH1 0x00
0x0127 SHA3
0x0128 DUP2
0x0129 SWAP1
0x012a SSTORE
0x012b POP
0x012c JUMPDEST
0x012d POP
0x012e JUMP
        jumpdest: 0038

0x0038 JUMPDEST
0x0039 STOP
```

Fig. A·2 Control flow graph obtained from RA for create-based re-entrancy attacks.

Yuichiro Chinen received his M.E. degree from Graduate School of Information Science and Technology in Osaka University, Japan, in 2020. His research interests include blockchains. He has recently joined the M.S. course in the Graduate School of Information Science and Technology in Osaka University, Japan. His research interests include machine learning and information security.

Naoto Yanai received his B.E. degree from The National Institution of Academic Degrees and University Evaluation, Japan, in 2009, his M.S. Eng. from Graduate School of Systems and Information Engineering, University of Tsukuba, Japan, in 2011, and his Ph.D. degree in Engineering from Graduate School of Systems and Information Engineering, University of Tsukuba, Japan, in 2014. He is an assistant professor at Osaka University, Japan. His research area is information security.

Jason Paul Cruz received his B.S. degree in Electronics and Communications Engineering and M.S. degree in Electronics Engineering from the Ateneo de Manila University, Quezon City, Philippines, in 2009 and 2011, respectively, and his Ph.D. degree in Engineering from the Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan in 2017. He is currently a Specially Appointed Assistant Professor at Osaka University, Osaka, Japan. His current research interests include role-based access control, blockchain technology, hash functions and algorithms, privacy-preserving cryptography, and Android programming.

Shingo Okamura received his B.E., M.E., and Ph.D. degrees from Graduate School of Information Science and Technology in Osaka University in 2000, 2002, and 2005, respectively. Since 2005, he has worked for Osaka University. In 2008, he joined National Institute of Technology, Nara College. Currently, he is an associate professor at the college. His research interests include cryptographic protocols and cyber security. He is a member of IEICE, IPSJ, IEEJ, ACM, IEEE, and IACR.