ReGuard: Finding Reentrancy Bugs in Smart Contracts

Chao Liu*
Peking University
Beijing, China
liuchao_cs@pku.edu.cn

Han Liu*
Chieftin Lab
Shenzhen, China
Tsinghua University
Beijing, China
liuhan2017@mail.tsinghua.edu.cn

Zhao Cao
Chieftin Lab
Shenzhen, China

Zhong Chen†
Peking University
Beijing, China
zhongchen@pku.edu.cn

Bangdao Chen
Chieftin Lab
Shenzhen, China

Bill Roscoe
Chieftin Lab
Shenzhen, China
University of Oxford
Oxford, UK

ABSTRACT

Smart contracts enabled a new way to perform cryptocurrency transactions over blockchains. While this emerging technique introduces free-of-conflicts and transparency, smart contract itself is vulnerable. As a special form of computer program, smart contract can hardly get rid of bugs. Even worse, an exploitable security bug can lead to catastrophic consequences, *e.g.*, loss of cryptocurrency/money. In this demo paper, we focus on the most common type of security bugs in smart contracts, *i.e.*, reentrancy bug, which caused the famous DAO attack with a loss of 60 million US dollars. We presented ReGuard, an fuzzing-based analyzer to automatically detect reentrancy bugs in Ethereum smart contracts. Specifically, ReGuard performs fuzz testing on smart contracts by iteratively generating random but diverse transactions. Based on the runtime traces, ReGuard further dynamically identifies reentrancy vulnerabilities. In the preliminary evaluation, we have analyzed 5 existing Ethereum contracts. ReGuard automatically flagged 7 previously unreported reentrancy bugs. A demo video of ReGuard is at https://youtu.be/XxJ3_-cmUiY.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*;

KEYWORDS

Smart contract, reentrancy bug, dynamic analysis

ACM Reference Format:
Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: Finding Reentrancy Bugs in Smart Contracts. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183495

1 INTRODUCTION

The passed several years have been witnessing a rapid development of blockchain applications since Bitcoin was first introduced in 2009 [12]. Unlike peer-to-peer payments of Bitcoin, blockchain recently enabled a new way to perform more complicated transactions which are encoded in *smart contracts*. Generally, smart contracts are a type of agreements among distributed and mutually distrusting participants, which are automatically made effective via the blockchain consensus mechanism. Additionally, a smart contract can be viewed as a special form of computer program that runs over a blockchain. Users can invoke specific contracts by sending transactions to a unique address, *i.e.*, an identifier of a smart contract. Typical transactions include exchange of cryptocurrencies (*e.g.*, ether), updates of blockchain status *etc.*. While smart contracts provide conflict-free and transparent way to perform real-world transactions, they are easily exposed to various security attacks. Even worse, such attacks on exploitable vulnerabilities may lead to catastrophic consequences, *e.g.*, loss of money, financial disorder *etc.*. We have investigated the known attacks in the context of smart contracts and focused on the *Reentrancy* bug in this demo paper. Specifically, reentrancy bugs refer to reentrant function calls which divert money in an unexpected way. Next, we briefly explain this type of bug in Figure 1 using a simplified version of the infamous the DAO attack [14], which causes a loss of 60 million US dollars.

```
1  contract Victim {
2    bool flag = false;
3    function withdraw() {
4      if (flag || !msg.sender.call.value(1 wei)()) throw;
5      flag = true;
6    }
7  }
8  contract Attacker {
9    uint count = 0;
10   function() payable {
11     if(++count < 10) Victim(msg.sender).withdraw();
12   }
13 }
```

Figure 1: A simplified version of the DAO attack.

The example code was written in Solidity (a programming language for Ethereum smart contracts) and has two contracts `Victim` and `Attacker`. The `Victim` contract has a `withdraw` function which sends ether (the Ethereum cryptocurrency) from the callee to the caller (line 4). Once the transfer is finished, the caller's fallback function will be invoked. The DAO attack is realized in the following scenario: (1) the attacker initiates a transaction by calling

*Both are first authors and contribute equally to the paper.
†Correspondence author

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183495

```
1  contract Simple {
2    bool public flag = false;
3    uint public amount = 0;
4
5    function update(uint _amount) {
6      flag = true;
7      amount = _amount;
8    }
9
10   function withdraw() payable {
11     if(flag)
12       msg.sender.call.value(amount);
13   }
14 }
```

(a) Solidity smart contract

```
1  class Simple final : public Address {
2    struct State {
3      solidity::bool flag_ = false;
4      solidity::uint amount_ = 0;
5    };
6    State current, restore;
7    void update(solidity::uint _amount,
8                const Message& msg) {
9      current.flag_ = true;
10     current.amount_ = _amount;
11   }
12   void withdraw(const Message& msg) {
13     if(current.flag_)
14       msg.sender()->fallback_call(
15          Message(this, current.amount_));
16   }}
```

(b) C++ smart contract

```
 from: 0xca35b7
   to: Simple
input: {update: 10}

 from: 0xca35b7
   to: Simple
input: {withdraw: []}

 from: 0xca35b7
   to: Simple
input: {update: 20}
```

(c) Transaction

Figure 2: Source-to-source smart contract transformation

`withdraw` function of `Victim`; (2) `Victim` transfers the money (line 4) and calls the fallback function (line 10-12); (3) The fallback function recursively calls the `withdraw` function again, *i.e.*, reentrancy; (4) Within an iteration bound, extra ether will be transfered multiple times. As described above, the reentrancy bug is easy to exploit on deployed blockchain contracts and has already become one of the most common attacks in practice. Unfortunately, we have not only relatively little knowledge on how to capture such bugs `automatically` and `accurately`, but also very limited tool support as well. From the practical perspective, detecting reentrancy bugs in smart contracts is facing the following challenges.

Challenge 1: Cover Transaction Scenarios. The execution of smart contracts may vary across different transactions. However, enumerating all possible scenarios is not only expensive in practice, but sometimes infeasible *w.r.t.* unbounded transaction space.

Challenge 2: Encode Reentrancy Property. The reentrancy property lacks an accurate definition in the context of smart contracts. Embedding simple patterns in the detection will introduce annoying false positives, while strict patterns miss bugs.

ReGuard Solution. To address the challenges, we have developed ReGuard for detecting reentrancy bugs in smart contracts. ReGuard introduces a translation-based framework from specific smart contract programming languages to C++. In the detection, ReGuard extends well-designed fuzzing engine (*e.g.*, AFL [1] and LibFuzzer [2]) by generating *random transactions*. During the runtime, ReGuard records the critical execution traces and feeds them into a *reentrancy automata* to identify potential bugs. We have instantiated ReGuard for Ethereum, and found 7 reentrancy bugs in 5 deployed contracts. A video of ReGuard is at https://youtu.be/XxJ3_-cmUiY.

2 OVERVIEW

The architecture of ReGuard is shown in Figure 3. In a high level picture, ReGuard takes as input the code of a smart contract. Specifically, both source code and binary code are supported. In terms of the output, ReGuard generates a bug report which is a collection of reentrancy bugs found in the smart contract under analysis. For each bug reported, ReGuard identifies the corresponding source location and an attack transaction which triggers the bug[1].

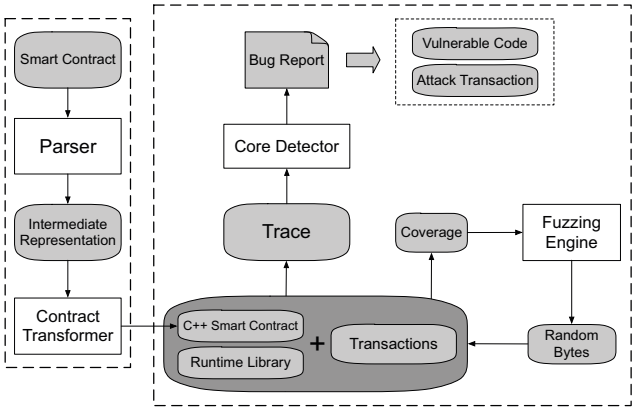

Figure 3: The architecture of ReGuard.

Next, we introduce the workflow of ReGuard. Given the code of a smart contract, ReGuard parses it into an intermediate representation (IR). For source code contract, the IR is an abstract syntax tree (AST). For binary code contract, we use a control-flow graph as its IR. Then, ReGuard performs a source-to-source transformation from IR to C++ in the *Contract Transformer*. The transformed C++ contract keeps the original behavior. On the other side, ReGuard embeds a fuzzing engine which will iteratively generate random bytes using runtime coverage information as a feedback. ReGuard interprets the bytes as transaction requests sent to the contract. With the C++ smart contract, transactions and a runtime library combined, ReGuard executes the contract and dumps runtime trace of analysis-relevant operations, *e.g.*, function call and return, access to storage and blockchain parameters, branching operations *etc.*. The trace is then passed to the *Core Detector* for reentrancy analysis. At last, bugs found are reported. Next, we describe three key modules of ReGuard, namely, *Contract Transformer*, *Fuzzing Engine* and *Core Detector*.

[1] ReGuard shows only one feasible attack transaction.

2.1 Contract Transformer

As aforementioned, ReGuard performs source-to-source transformation from IR to C++. The transformation is realized in the *Contract Transformer*. We take the AST IR as an example to explain the transformation, which conceptually amounts to traversing the AST and generating C++ code for specific types of AST nodes. Considering the code in Figure 2a, the contract written in Solidity [7] declares and initializes two storage variables `flag` and `amount`. Additionally, it defines two callable functions, *i.e.*, `update` which updates values of storage and `withdraw` which transfers ether to any calling participant.

Figure 2b shows the transformed C++ contract. Specifically, the two functions `update` and `withdraw` are transformed accordingly. Besides original function arguments, a `Message` object is explicitly passed into C++ functions to carry transaction information, *e.g.*, the sender's address and ether included *etc.*. At line 14-15, the `fallback_call` function is invoked to simulate operations of fallback function in the calling address. In sum, the transformation keeps the original behavior and interfaces to the fuzzing engine for transaction generation.

2.2 Fuzzing Engine

The execution of a smart contract is highly transaction-dependent. That is, different transactions may lead to different contract states, in which reentrancy bugs are likely to manifest. In order to explore the state space of smart contract as much as possible, ReGuard takes the advantage of well-designed fuzzing engines to generate random transactions and cover different execution scenarios, as shown in Figure 4.

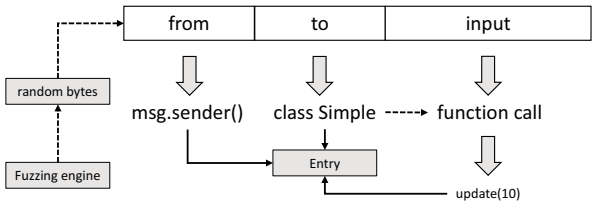

Figure 4: Interpret random bytes into transactions.

The ReGuard tool leverages a fuzzing engine to iteratively produce random bytes, which are then interpreted as a collection of triples ⟨*from*, *to*, *input*⟩. The *from* field specifies the sender address of the message. The *to* field indicates to which contract (C++ class) the transaction will be sent (in this case `Simple` as in Figure 2a). The *input* field generates a random call to a function defined in the contract (in this case `update` with argument 10 as in Figure 2a). Next, the triple will be embedded in a main entry which executes the C++ smart contract. During execution, runtime coverage is recorded to optimize byte generation. Considering the case in Figure 2c, the fuzzing engine generates three transactions, which are all from the same sender to the contract `Simple`. and produce a call sequence `{update(10), withdraw(), update(20)}`.

2.3 Core Detector

As the fuzzing engine runs, ReGuard executes a smart contract with different transactions and dumps runtime trace. The trace is analyzed by the *Core Detector* to identify reentrancy bugs. Specifically, the detection is realized via parsing the trace in a *reentrancy automata*, which is shown in Figure 5.

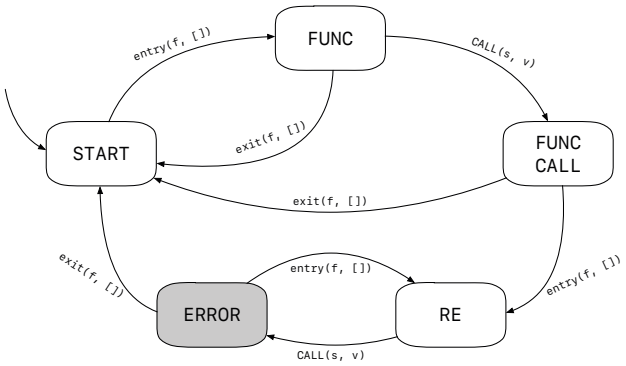

Figure 5: Reentrancy Automata. `entry/exit`: function entry and exit. `CALL`: the `CALL` opcode of Ethereum virtual machine.

The reentrancy automata is a form of finite state machine (FSM), where states indicate the occurrence of reentrancy bugs and transitions are recorded operations in a trace. In total, five states are specified, *i.e.*, `START`, `FUNC`, `FUNC CALL`, `RE` and `ERROR`. `START` is the entry point of the automata. `ERROR` indicates that a reentrancy bug occurs. `FUNC` models a call to a function and `FUNC CALL` further refines the call with a ether transfer operation, *i.e.*, the `CALL` instruction. `RE` flags that the execution introduces a reentrant function call. Once the call tries to transfer ether again, the automata goes to the `ERROR` state and reports the bug. At runtime, ReGuard records the current state in the automata and identify bugs on the fly.

3 USAGE

We've deployed ReGuard as a web service. Figure 6 showed a screenshot of ReGuard. To analyze a smart contract, ReGuard adopts a "one-click" design. Users only need to type the contract code in the browser, and ReGuard will take care of everything else, *i.e.*, syntax check, code transformation, fuzzing, and bug detection.

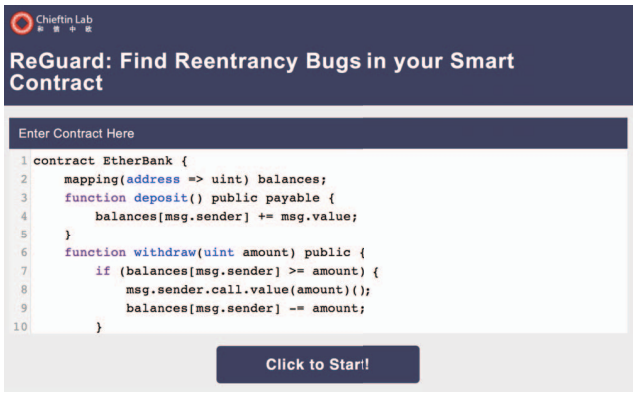

Figure 6: The ReGuard screenshot.

Taking the simple smart contract in Figure 6 as a running example, we briefly describe the usage of ReGuard. By clicking the "Start" button, ReGuard is launched for reentrancy analysis. When the detection is done, a bug report is generated at bottom as shown in Figure 7. In this case, a reentrancy bug is highlighted at line 8, *i.e.*, ether transfer operation using the `call.value()` API in the `withdraw` function.

Figure 7: Analysis result of `EtherBank`.

Table 1: Results of reentrancy bug detection. FP: false positive. FN: false negative. $*$ means the analysis did not finish within the time limit.

Contract	LOC	Oyente	ReGuard
TokenSender	129	(FP) 2	0
UGToken	226	*	2
E4Token	871	(FN) 0	1
DAO	1240	3	3
RoT	1242	(FN) 0	1

In addition to the error reported, ReGuard further shows the call stack information which explains the reentrancy bug. Specifically, when line 8 is executed, the fallback function of the message sender will be called. In the fallback function, the sender calls back to the `withdraw` function again. Since line 8 does not introduce any protection logic, the reentrant call still passes and extra ether gets transferred. The call stack information enables developers to better understand the bug and design fix patches.

4 PRELIMINARY EVALUATION

We have conducted preliminary evaluation of ReGuard on 5 modified smart contracts from `Etherscan`[2]. All the experiments were performed on a Ubuntu 16.04 laptop with dual i7 CPU@2.60 GHz 2.81GHz, and 16GB memory. We used Oyente [11] as a comparison to investigate differences between fuzzing (ReGuard) and symbolic execution (Oyente) based techniques. While Oyente allows checking on a set of bug patterns, we focused on reentrancy bugs in this setting. The analysis time limit is set to 20 minutes. The results are shown in Table 1.

For all collected contracts, ReGuard finished detection in 20 minutes. While Oyente provided better coverage guarantee of testing smart contracts, ReGuard managed to avoid both false positives and negatives in finding bugs (*i.e.*, line 1, 3, 5 in Table 1).

5 RELATED WORK

The security of smart contracts has been attracting research concerns as blockchain applications become popular [3, 5, 8, 10, 13]. Delmolino *et al.* show that even a rather simple smart contract (*e.g.*, "Rock, Paper, Scissors") may lead to various kinds of flaws [6]. Based on formal methods, Hirai reasoned smart contract using Isabelle/HOL [9]. Bhargavan *et al.* introduced a framework which translates smart contracts to F* language programs for formal verification [4]. The major limitation of such techniques is that the analysis is semi-automatic, which involves manual work, thus is insufficient for large-scale application. To the best of our knowledge, the most related work is from Luu *et al.* [11], who introduced four types of security bugs in Ethereum smart contracts, *i.e.*, transaction-ordering dependence, timestamp dependence, mishandled exceptions, reentrancy vulnerability, and developed the Oyente analyzer. Oyente performs symbolic execution on contract functions and flags bugs based on simple patterns. Compared to Oyente, ReGuard focus more on generating diverse transactions and refining the bug patterns. Conceptually, ReGuard can serve as a complementary to Oyente (Oyente is more aware of the contract control flow but ReGuard is more lightweight and suitable for complex contracts) and be combined together.

6 CONCLUSION

In this demo paper, we presented ReGuard, a dynamic analyzer for reentrancy bugs in smart contracts. ReGuard leverages fuzzing based techniques to generate random and diverse blockchain transactions as possible attacks. Then, ReGuard performs bug detection via runtime trace analysis. We have instantiated ReGuard for Ethereum contracts as a web service. On five modified smart contracts, ReGuard identified 7 bugs and avoided introducing both false positives and negatives.

ACKNOWLEDGMENTS

The authors would like to thank Thomas Gibson-Robinson, Philippa J. Hopcroft and Youcheng Sun for their help in preparing the tool. This work is supported by National Natural Science Foundation of China under the grant No.: 61672060 and China Postdoctoral Science Foundation under the grant No.: 2017M620785.

REFERENCES

[1] 2017. AFL. http://lcamtuf.coredump.cx/afl/. (2017).
[2] 2017. LibFuzzer. llvm.org/docs/LibFuzzer.html. (2017).
[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *International Conference on Principles of Security and Trust*. Springer, 164–186.
[4] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, A Rastogi, T Sibut-Pinote, N Swamy, and S Zanella-Beguelin. 2016. Formal verification of smart contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS'16*. 91–96.
[5] Michael Coblenz. 2017. Obsidian: A safer Blockchain programming language. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 97–99.
[6] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*. Springer, 79–94.
[7] Ethereum. 2017. Solidity — Solidity 0.4.19 documentation. (2017). https://solidity.readthedocs.io/en/develop/
[8] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. 2017. *KEVM: A Complete Semantics of the Ethereum Virtual Machine*. Technical Report.
[9] Yoichi Hirai. 2016. Formal verification of Deed contract in Ethereum name service. (2016).
[10] Yoichi Hirai. 2017. Defining the ethereum virtual machine for interactive theorem provers. WSTC17. In *International Conference on Financial Cryptography and Data Security*.
[11] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.
[12] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Consulted* (2008).
[13] Jack Pettersson and Robert Edström. 2015. Safer smart contracts through type-driven development. (2015).
[14] WIRED. 2016. A $50 MILLION HACK JUST SHOWED THAT THE DAO WAS ALL TOO HUMAN. (2016). https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/

[2] https://etherscan.io