SMARTSHIELD: Automatic Smart Contract Protection Made Easy

Yuyao Zhang*, Siqi Ma†(✉), Juanru Li*(✉), Kailai Li*, Surya Nepal†, and Dawu Gu*
*Shanghai Jiao Tong University, Shanghai, China
{breakingpoint, jarod, silence_916, dwgu}@sjtu.edu.cn
†Data61, CSIRO, Sydney, Australia
{siqi.ma, surya.nepal}@csiro.au

***Abstract*—The immutable feature of blockchain determines that traditional security response mechanisms (e.g., code patching) must change to remedy insecure smart contracts. The only proper way to protect a smart contract is to fix potential risks in its code before it is deployed to the blockchain. However, existing tools for smart contract security analysis focus on the detection of bugs but seldom consider the code fix issues. Meanwhile, it is often time-consuming and error-prone for a developer to understand and fix flawed code manually. In this paper we propose SMARTSHIELD, a bytecode rectification system, to fix three typical security-related bugs (i.e., *state changes after external calls*, *missing checks for out-of-bound arithmetic operations*, and *missing checks for failing external calls*) in smart contracts automatically and help developers release secure contracts. Moreover, SMARTSHIELD guarantees that the rectified contract is not only immune to certain attacks but also gas-friendly (i.e., a slightly increase of gas cost). To evaluate the effectiveness and efficiency of SMARTSHIELD, we applied it to 28,621 real-world buggy contracts on *Ethereum* blockchain (as of January 2nd 2019). Experiment results demonstrated that among 95,502 insecure cases in those contracts, 87,346 (91.5%) of them were automatically fixed by SMARTSHIELD. A following test with both program analysis and real-world exploits further testified that the rectified contracts were secure against common attacks. Moreover, the rectification only introduced a 0.2% gas increment for each contract on average.**

***Index Terms*—*Ethereum* blockchain, Smart contract, Automated bug fix, Bytecode rectification**

I. INTRODUCTION

Since the birth of *Ethereum* in 2015, various severe bugs in smart contracts[1] have been exploited, resulting in a loss of tens of millions USD worth of Ether [1]. To remedy this situation, two aspects of efforts have been made. On the one hand, a wide variety of approaches have been proposed to identify bugs in smart contracts. Some [2]–[5] utilize symbolic execution to discover potential security bugs, while others [6]–[9] leverage pattern matching and fuzzing to detect real-world security bugs in smart contracts. On the other hand, a large number of programming guides with best-practice recommendations [10]–[13] have been released to help developers avoid common pitfalls.

Despite the aforementioned efforts are made, a majority of smart contracts on the *Ethereum* blockchain are still developed without following the best-practice recommendations. Severe threats happened on smart contracts [14] and a large number of bugs are reported every day in which 25% of them are critical bugs [15]. We argue that most existing tools (e.g., *Securify* [6], *Osiris* [16], and *Mythril* [17]) aiming at detecting smart contract bugs are insufficient: they only report where bugs locate instead of helping developers fix the buggy code. Although a bug is detected, manually bug fixing is not only error-prone, but also time-consuming because 1) numerous suggestions with incorrect implementations are posted online [18]; 2) developers have to learn dependencies in each contract before fixing to avoid side effects. As a result, an automated code fix approach is expected to help developer deploy bug-free contracts **before** the deployment. Otherwise, bug detection tools only benefit attackers to exploit buggy contracts while **deployed smart contracts are not allowed to be updated on blockchain**.

As typical security-related bugs have certain insecure patterns, it is feasible to fix these bugs by revising these insecure patterns [19]. To help developers fix security-related bugs automatically, a variety of automatic program repair techniques have been proposed [20]. These repair techniques, targeting on different program languages such as Java [21], [22] and C [23], [24], rely on various mutators such as replacing field accesses, method invocations. However, the above techniques are unable to be applied to rectify insecure smart contracts. Comparing with Ethereum Virtual Machine (EVM) bytecode, source code and bytecode of the high-level programs are more semantically understandable. On the contrary, EVM bytecode is designed for efficient execution and compacting program representation. It not only omits notions of structs and objects, but also ignores the concept of method. Operations targeting on these structures become useless.

Hence, we make a first step towards a general-purpose smart contract protection against attacks exploiting insecure smart contracts. We design and implement an automated smart contract rectification system, SMARTSHIELD. It fixes insecure cases in each vulnerable smart contract to secure the EVM bytecode of the contract for the final deployment. SMARTSHIELD takes three steps to fix insecure cases in each smart contract: **1)** it conducts a semantic-preserving code transformation to assure that only the insecure code patterns are revised, and the functionalities of the irrelevant functions are consistent; **2)** it follows the gas-friendly requirement by

[1]In this paper we focus on smart contracts on *Ethereum* platform. We use the concept *smart contract* to denote *Ethereum Smart Contract*.

adopting heuristics to optimize gas consumption[2] and thus the increments of gas consumption are restricted to an acceptable level; **3)** it recognizes a particular insecure case whose fix may lead to side effects and sends its rectification suggestion back to developers, which help them rectify contracts as well as improve the code quality.

To evaluate SMARTSHIELD, we first built a dataset with labeled insecure cases. We collected 28,621 contracts with insecure code from 2,214,409 real-world smart contracts (from the *Ethereum Mainnet*, ETH). In total, these insecure contracts contains 95,502 insecure cases. Then, we utilized SMARTSHIELD to fix the insecure cases. 87,346 (91.5%) insecure cases were fixed and the secure version of 25,060 (87.6%) contracts were generated. To testify whether these rectified contracts are secure, we simulate the attacker with both program analysis tools and real world exploits. We leveraged three state-of-the-art smart contract vulnerability detection tools (i.e., *Securify* [6], *Osiris* [16], and *Mythril* [17]) to find bugs in the rectified contracts, and replayed 55 exploits that had caused serious impacts against existing contracts. The results showed that contracts rectified by SMARTSHIELD were bug-free and thwart all exploits. We also replayed historical transactions to validate the normal functionalities in the rectified contracts, and found only 60 (0.24%) rectified contracts encountered execution inconsistency Additionally, the historical transaction replay demonstrated that the increment of gas consumption was only 0.2% on average for each rectified contract. Our experimental results indicated that SMARTSHIELD is effective and efficient in protecting diverse real-world insecure contracts.

II. Motivation

Targeting on smart contracts, we observe that a large portion of insecure cases containing common patterns. It indicates that these bugs can be identified and fixed through a unified approach. In the following, we first discuss three representative security-related bugs in smart contracts and summarize their insecure code patterns. Then, we present a smart contract rectification approach, which generates secure EVM bytecode by fixing these bugs. Through the rectification approach, insecure contracts are automatically rectified before their deployment even though the insecure cases are mistakenly introduced by developers. The security level of the entire ecosystem is significantly improved.

A. Insecure Code Patterns in Smart Contracts

In this paper, we discuss three typical insecure code patterns: *state changes after external calls*, *missing checks for out-of-bound arithmetic operations*, and *missing checks for failing external calls*. According to both security requirements outlined in official *Solidity* document [10] and other proposed best practices of smart contracts [11]–[13], these patterns are the most critical security risks to smart contracts consistently. We present the details below.

[2] *Gas* indicates the fee that needs to be paid to the *Ethereum* network in order to conduct a transaction or execute a smart contract.

State Changes after External Calls. This insecure code pattern indicates the scenario that a state variable (i.e., a variable stored in the storage) is updated after an external function call. It may result in a **re-entrancy bug** under particular circumstances. Actually, one of the most notorious attacks, the *DAO* attack, occurred on June 18th, 2016 and led to the hard fork of the *Ethereum* blockchain, exactly exploited such insecure code in the *DAO* contract [25].

```
1 mapping (address => uint) public userBalances;
2 ...
3 function withdrawBalance(uint amountToWithdraw) public {
4     require(userBalances[msg.sender] >= amountToWithDraw);
5 +   userBalances[msg.sender] -= amountToWithdraw;
6     msg.sender.call.value(amountToWithdraw)();
7 -   userBalances[msg.sender] -= amountToWithdraw;
8 }
```

Fig. 1: An example of state changes after external calls

In Figure 1, we give a concrete example of *state changes after external calls*. The user is allowed to withdraw funds by calling function *withdrawBalance* declared from line 3 to line 8, in which function *call.value(.)* at line 6 calls a function in another contract. Suppose that an attacker uses a malicious contract to call function *withdrawBalance* declared in an insecure contract, it calls back to the malicious contract by executing function *call.value(.)* at line 6. Simultaneously, the insecure contract hands over control to the malicious contract as well as sending *Ether*. Because the control is handed over, the insecure contract then waits until the external call returns. Although *Ether* has been sent to the malicious contract, the state variable *userBalances* at line 7 is not updated yet. Therefore, the malicious contract can bypass the validity check declared at line 4 and launch the re-entrancy attack to ask for fund withdrawal again.

```
1  uint public lockTime = now + 1 weeks;
2  address public user;
3  ...
4  function increaseLockTime(uint timeToIncrease) public {
5      require(msg.sender == user);
6  +   require(lockTime + timeToIncrease >= lockTime);
7      lockTime += timeToIncrease;
8  }
9  ...
10 function withdrawFunds() public {
11     require(now > lockTime);
12     user.transfer(address(this).balance);
13 }
```

Fig. 2: An example of missing checks for out-of-bound arithmetic operations

Missing Checks for Out-of-Bound Arithmetic Operations. This insecure code pattern depicts the scenario that an arithmetic statement is operated without checking the data validity in advance. If an out-of-bound result is produced by the arithmetic operation, an **arithmetic bug**, such as integer overflow and underflow, will be caused. In 2018, an integer overflow bug has affected more than a dozen of ERC20

contracts and directly led to withdrawal and trading suspension of BeautyChain (BEC) token [26].

The sample code containing a *missing checks for out-of-bound arithmetic operations* pattern is shown in Figure 2. According to the statement at line 11 in function *withdrawFunds*, user's fund is locked for a week before it can be withdrawn. Optionally, the lock time (i.e., the unsigned integer variable *lockTime*) can also be increased by calling function *increaseLockTime*. However, the time increment relies on the parameter *timeToIncrease*, which is given by a user. A malicious user can provide a large number for the addition operation to produce an overflow at line 7, and bypass the statement of time check at line 11.

Missing Checks for Failing External Calls. This insecure pattern specifies that the return value is not being checked after calling a function in an external contract or after sending *Ethers*. If the contract invokes an function in another contract by using a low-level operation (e.g., *call*, *send*), instead of a high-level *transfer*. Exceptions thrown by the callee contract cannot be propagated to the caller contract. Instead, a boolean value — *false* is returned. If the caller contract does not check the return value, an **unchecked return value bug** may be caused.

```
1  bool public payedOut = false;
2  address public winner;
3  uint public bonus;
4  ...
5  function sendToWinner() public {
6      require(!payedOut && msg.sender == winner);
7  -   msg.sender.send(bonus);
8  +   require(msg.sender.send(bonus));
9      payedOut = true;
10 }
```

Fig. 3: An example of missing checks for failing external calls

In the contract listed in Figure 3, the *sendToWinner* function is used for winners to claim bonus. The contract sets the state of payment (*payedOut*) as *true* at line 9 after the execution of sending bonus by default, although the operation *send(.)* at line 7 may return false. However, the contract still sets *payedOut* as true even the bonus is failed to be sent to the winner. As a result, this contract can never send the bonus afterwards.

B. Automatic Rectification

We argue that insecure code patterns mentioned in Section II-A can be revised through certain code analysis and code transformation. To implement this target and thus protect smart contracts against corresponding attacks, we propose an approach to automatically rectify insecure smart contracts, which generates correct and optimized contract bytecode.

1) Insecure Pattern Revise: Following the best practice of checks-effects-interactions [27], we revise the three insecure code patterns as below.

State Changes after External Calls. The insecure pattern, *state changes after external calls*, is caused by an incorrect execution sequence of *state change* and *external call*. It can be revised by moving all state changes to the front of an external function call. Consider the example in Figure 1, the insecure code can be fixed by moving the statement of the state change at line 7 to the front of the statement of the external function call at line 6.

Missing Checks for Out-of-Bound Arithmetic Operations. The insecure pattern of *missing checks for out-of-bound arithmetic operations* can be resolved by inserting a boundary check to an arithmetic operation. A concrete example in Figure 2 at line 6 demonstrates how to check such out-of-bound exception. As each type of the integer variables in *Solidity* [28] is restricted in a certain range, the insecure pattern are revised by inserting a validity check before the arithmetic operation at line 7 to examine whether the result is out of range.

Missing Checks for Failing External Calls. To address *missing checks for failing external calls*, a validity check must be provided to guard the return of a function call that invokes a function in an external contract. In Figure 3, the risk is mitigated by inserting a validity check at line 7 after the external function invoking.

2) Bytecode Generating: According to the revise suggestions for insecure patterns, SMARTSHIELD applies the following techniques to generate rectified smart contracts.

Bytecode-Level Program Analysis. Different from natural languages, the semantic information at the bytecode level is unified. To obtain the semantic information, a bytecode-level program analysis (refer to Definition III-A) is required. Such semantic information is necessary for further code transformation and secure bytecode generation (Section III-A).

Semantic-Preserving Code Transformation. To guarantee that the insecure code fix does not affect other original functionalities, our rectification applies a semantic-preserving transformation technique to compile each insecure contract to a secure bytecode version. Based on the extracted bytecode-level semantic information, the semantic-preserving transformation proves that only the insecure code snippet is adjusted and functionalities irrelevant to the insecure case are not affected (Section III-B).

Gas Optimization. An important feature of the smart contract is that each instruction consumes a certain amount of *gas* during the contract execution. Deploying a contract costs 200 units of gas per byte [29]. Hence, we should avoid unnecessary instructions while generating bytecode. To implement a gas-friendly bytecode generation, our rectification introduces several optimization policies to guarantee a minimum gas consumption during the contract execution (Section III-B).

III. SMARTSHIELD

In this section, we introduce our automated smart contract rectification system, SMARTSHIELD, in detail. Designed to automatically fix insecure cases with typical insecure code patterns in smart contracts, SMARTSHIELD takes a smart contract as input and outputs a secure EVM bytecode without any of the three insecure code patterns (i.e., *state changes*

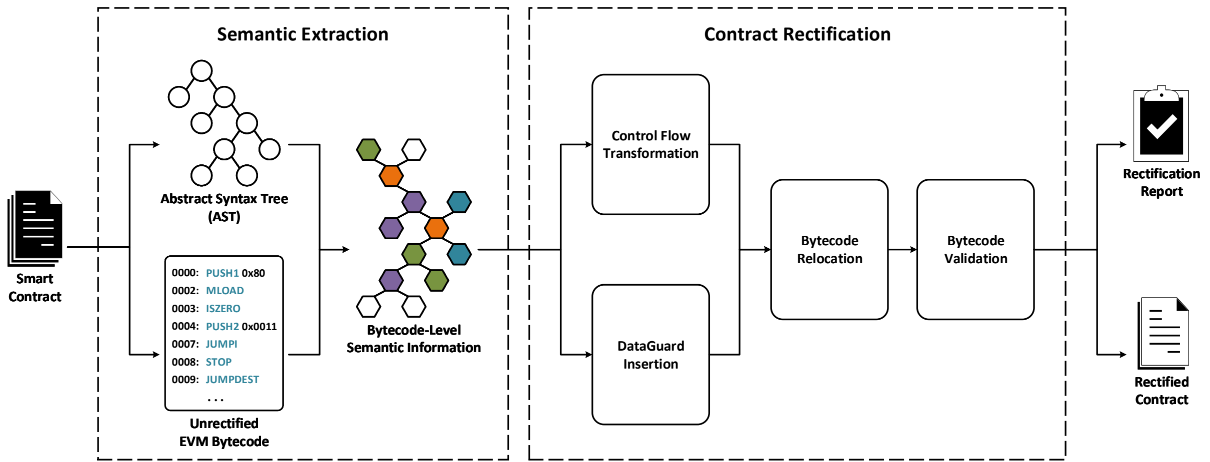


Fig. 4: Workflow of SMARTSHIELD

after external calls, *missing checks for out-of-bound arithmetic operations*, and *missing checks for failing external calls*). Figure 4 depicts the workflow of SMARTSHIELD, which contains two phases:

1) **Semantic extraction:** SMARTSHIELD analyzes both the abstract syntax tree (AST) and the unrectified EVM bytecode of each contract to extract its bytecode-level semantic information.
2) **Contract rectification:** Given the extracted bytecode-level semantic information, SMARTSHIELD fixes insecure control flows and data operations through *control flow transformation* and *DataGuard insertion*, respectively.

Finally, SMARTSHIELD generates the rectified EVM bytecode as well as a rectification report to the developer.

A. Semantic Extraction

Definition. *Bytecode-level semantic information in a smart contract refers to control and data dependencies among instructions in EVM bytecode.*

Since the EVM is a stack-based virtual machine and its bytecode is compact and highly optimized, information collected from bytecode level may not be complete. To obtain precise bytecode-level semantic information, SMARTSHIELD analyzes the abstract syntax tree (AST) of each contract combined with its unrectified EVM bytecode.

SMARTSHIELD parses the source code of each contract and builds an AST to derive control flows among statements and data flows among variables. It first generates a control flow graph (CFG) to extract control flows. Each node in the CFG is a statement and a directed edge is added between two nodes. The direction of an edge follows the execution sequence of the two statements, which represents their control flow. Refer to CFG, SMARTSHIELD further extracts data flows. In advance, SMARTSHIELD collects the following variable information from the AST: (1) variable types such as local variables, state variables, parameters; (2) data types such as *bool*, *uint32*, *address*; (3) source maps (i.e., a mapping between a node in the AST and its original statement placed in the source code). According to the variable information and CFG, SMARTSHIELD performs data-flow analysis and determines dependencies among variables.

Nonetheless, the control and data flows extracted from the source code cannot be applied on bytecode directly. SMARTSHIELD regards these flows as basic references and further utilizes abstract execution to analyze EVM bytecode.

The abstract execution conducted by SMARTSHIELD abstractly emulates the execution of each contract bytecode and monitors the change of each execution state. During the execution, each execution state is represented as a 4-tuple (*Instruction*, *Stack[]*, *Memory[]*, *Storage[]*). *Instruction* represents the one that will be executed, and the remaining items describe the present elements in the stack, the memory, and the storage, respectively. In particular, the abstract execution observes the execution state changes from two perspectives, *control flow transfer* and *data manipulation*.

Control Flow Transfer. To monitor control flow transfer and infer control dependencies among instructions, SMARTSHIELD executes the following steps triggered by different instructions:

- **Execution halting:** SMARTSHIELD terminates the sequence of the current execution when instructions `STOP`, `RETURN`, `REVERT`, `INVALID`, or `SUICIDE`[3] are met.
- **Execution transfer:** SMARTSHIELD transfers the current execution after executing instructions `JUMP` and `JUMPI`. `JUMP` refers to an unconditional transfer; thus, SMARTSHIELD transfers the execution to the instruction at the target address, i.e., jump address. For `JUMPI`, SMARTSHIELD explores two execution sequences simultaneously because it transfers the execution under certain conditions. One branch follows the original execution sequence, and the other one executes instructions sequentially starting from the target address. SMARTSHIELD duplicates the current execution state when instruction `JUMPI` is encountered. To ensure that SMARTSHIELD terminates successfully, SMARTSHIELD terminates to

[3]Readers may refer to the formal specification of *Ethereum* [29] for obtaining more details on the instruction set of the EVM.

explore the execution sequence if an instruction halts the execution or the current state has been executed.

- **Unaltered execution:** SMARTSHIELD sequentially executes instructions not being included in the previous two categories.

Data Manipulation. SMARTSHIELD monitors how EVM bytecode manipulate data and obtains data dependencies. Since instructions in EVM bytecode operate on elements in either a stack or a memory, SMARTSHIELD separately emulates stack and memory manipulations as follows:

- **Stack manipulation:** If an instruction A pushes an operand v onto the stack, SMARTSHIELD labels v as "assigned by A". When an instruction B operates on the operand v from the stack, SMARTSHIELD checks the label of v and recognizes that B is data dependent on A.
- **Memory manipulation:** If an instruction A writes data to a specific memory region mem, SMARTSHIELD labels all addresses included in this memory region, $addr = addr_1, ..., addr_n$, as "assigned by A". For an instruction B who reads data from the memory region mem, SMARTSHIELD examines the label of $addr$ in mem and records that B is data dependent on A.

However, addresses to read or write may be unknown. For example, some addresses are computed depending on input data (i.e., calldata). We apply a conservative strategy to resolve this issue by analyzing all addresses over the entire memory. For an instruction writing to the memory, SMARTSHIELD labels all possible addresses that can be written by this instruction. Similarly, SMARTSHIELD extracts all possible addresses that can be read by an instruction and then creates data dependencies among these relevant instructions of reading and writing memory regions.

B. Contract Rectification

Based on the extracted semantic information, SMARTSHIELD adopts a semantic-preserving code transformation with two strategies, *control flow transformation* and *DataGuard insertion*, to fix insecure cases in a contract.

Control Flow Transformation. SMARTSHIELD adopts a control flow transformation to revise *state changes after external calls*. SMARTSHIELD adjusts the original control flow by moving the instruction `SSTORE` to the front of the instruction `CALL` if `SSTORE` is executed after `CALL` in a certain execution path. However, simply modifying the execution sequence of `SSTORE` and `CALL` may violate original dependencies among instructions. In response, SMARTSHIELD relies on the bytecode-level semantic information extracted in Section III-A to address this issue.

Given the semantic information among instructions, SMARTSHIELD first scans the instruction sequence between `SSTORE` and `CALL`, and constructs a set containing all instructions that `SSTORE` either directly or indirectly depends on. Then, SMARTSHIELD moves both `SSTORE` and instructions in the set to the front of `CALL`. Consider the insecure case containing *state changes after external calls* shown in

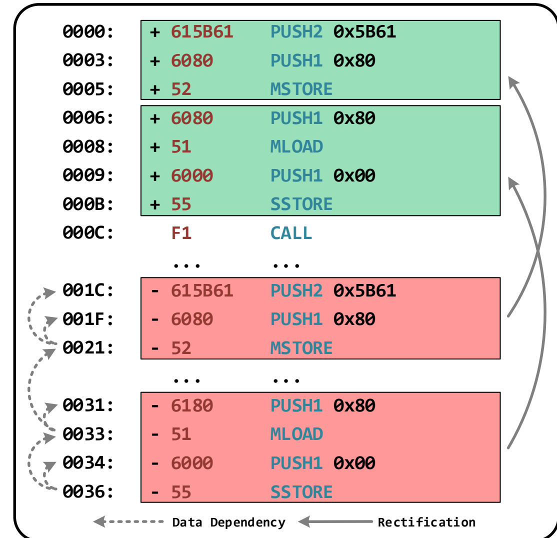


Fig. 5: An example of control flow transformation

Figure 5. `SSTORE` at address 0x36 conducts a state change after calling a function in an external contract made by `CALL` at address 0x0C. SMARTSHIELD first constructs a set consisting of all the predecessors of `SSTORE`, i.e., `PUSH1` at address 0x34, `MLOAD` at address 0x33, `PUSH1` at address 0x31, `MSTORE` at address 0x21, `PUSH1` at address 0x1F, and `PUSH2` at address 0x1C. Then, SMARTSHIELD moves those instructions together with `SSTORE` to fix the insecure case.

Note that the moved instructions may depend on the execution result of `CALL`. In such circumstance, the case with *state changes after external calls* pattern cannot be fixed due to the dependency conflict, and thus a manual refactoring of the contract is expected. Consequently, SMARTSHIELD just warns this case to the developer in the rectification report.

TABLE I: DATAGUARDS FOR OUT-OF-BOUND ARITHMETIC OPERATIONS AND FAILING EXTERNAL CALLS

Category	Instruction	Operation	DataGuard
Arithmetic ops	ADD	$a+b$	$a+b \geq a$
	SUB	$a-b$	$a \geq b$
	MUL	$a \times b$	$a \times b \div a = b$
External calls	CALL	$ret = a.call()$	$ret \neq 0$

DataGuard Insertion. SMARTSHIELD inserts specific DataGuards to fix insecure cases with *missing checks for out-of-bound arithmetic operations* and *missing checks for failing external calls*. A DataGuard is a sequence of instructions that performs certain data validity checks. SMARTSHIELD introduces four manually built secure functions, containing three secure arithmetic operations (i.e., secure operations to execute instructions `ADD`, `SUB`, and `MUL`) and a secure external call (`CALL`). Four DataGuards are illustrated in Table I. When SMARTSHIELD locates a potential insecure operation, it will utilize the corresponding DataGuard to check the result and preclude attacks.

If SMARTSHIELD adds multiple DataGuards for all insecure operations, the size of the EVM bytecode increases sig-

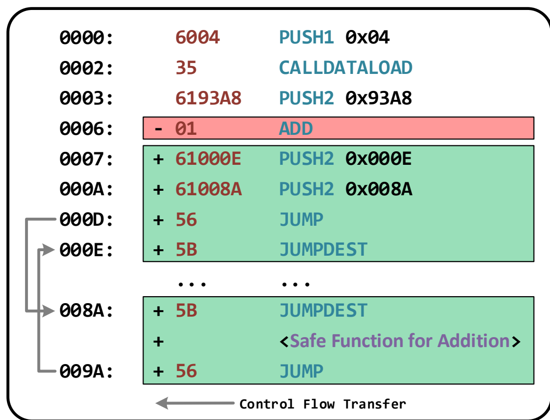


Fig. 6: An example of DataGuard insertion

nificantly. To achieve the goal of gas optimization, SMARTSHIELD first appends the secure functions to the end of the generated EVM bytecode. Then SMARTSHIELD replaces the insecure operation (e.g., an external call) by a secure function invocation. Taken the EVM bytecode snippet in Figure 6 as an example, `ADD` at address 0x06 performs an *add* operation without checking the validity of the input data, which might cause an out-of-bound issue. To fix this bug, SMARTSHIELD replaces `ADD` by invoking the appended secure `ADD` function (from address 0x8A to address 0x9A). Note that EVM bytecode instructions do not support operations of function calls and returns within a single contract. Therefore, SMARTSHIELD directly pushes the target address and the return address (code from address 0x07 to address 0x0E) to implement a function invocation.

C. Rectified Contract Generation

After the insecure cases are fixed, SMARTSHIELD refines and validates the rectified EVM bytecode, and then generates a rectification report.

Bytecode Relocation. As the contract rectification may change addresses of instructions, all unaligned target addresses of instructions `JUMP` and `JUMPI` are required to be updated to ensure the correct jump targets. In the EVM bytecode, the target address of `JUMP` or `JUMPI` is generally pushed onto the stack by a `PUSH` in advance. By analyzing the data dependencies of `JUMP` or `JUMPI`, SMARTSHIELD identifies the correlated `PUSH` instruction and updates the unaligned target address.

Bytecode Validation. Given a rectified bytecode, SMARTSHIELD validates whether the other irrelevant functions are affected. SMARTSHIELD first extracts the bytecode-level semantic information from the rectified contract. It then compares the execution sequences and data flows of the unmodified instructions with the original ones.

Rectification Report. Together with the rectified contract, SMARTSHIELD generates a report to the developer. For the successfully fixed cases, SMARTSHIELD records the concrete modifications and leverages source maps to reflect the modifications made on the AST. For those fixed cases that fail to be validated, SMARTSHIELD marks them as "unrectifiable". Fix suggestions are included in the rectification report for further manual verification or adjustments.

IV. Evaluation

In this section, we evaluate the effectiveness and efficiency of SMARTSHIELD. Specifically, we evaluate SMARTSHIELD by answering the following research questions:

RQ1: Scalability. As SMARTSHIELD is proposed to protect smart contracts automatically, how scalable is SMARTSHIELD in rectifying real-world smart contracts?

RQ2: Correctness. As SMARTSHIELD is designed to rectify insecure contracts, how effective and accurate is SMARTSHIELD in fixing insecure cases with the insecure patterns and assuring the functionality consistency between the rectified and the original contract?

RQ3: Cost. As SMARTSHIELD is introduced to help developers generate a secure and optimized smart contract, what is the additional cost of the contract generation?

TABLE II: Insecure cases in our dataset

Category	# of insecure cases	# of insecure contracts
CP.1	4,521	726
CP.2	80,825	25,470
CP.3	10,156	4,811
Total	95,502	28,621*

∗ Some contracts contain multiple insecure patterns.
CP.1: State Changes after External Calls
CP.2: Missing Checks for Out-of-Bound Arithmetic Ops
CP.3: Missing Checks for Failing External Calls

A. Experimental Setup

Dataset. To build a dataset of insecure contracts, we first collected real-world smart contracts with source code available on the *Ethereum Mainnet* (ETH). We used *geth* [30] to synchronize the *Ethereum* network by January 2nd, 2019 and downloaded a snapshot of the first 7,000,000 blocks containing 369,817,320 transactions. Since each contract is created through sending a relevant transaction to an empty address (i.e., address 0x0 on the blockchain), we thus distinguished 2,214,409 real-world contracts deployed on the *Ethereum* network. Then, we referred to the Ethereum blockchain explorer–*Etherscan* [31] to investigate the existence of source code for these collected contracts and found 52,179 contracts that have source code available online. With the source code, we labeled the insecure code patterns defined in Section II-A. We made use of state-of-the-art smart contract analysis tools (e.g., *Securify* [6], *Osiris* [16], *Mythril* [17], etc.) to help label insecure cases. By cross checking the results generated by various tools and manually inspecting the source code of controversial contracts, we finally built our dataset.

More specifically, our dataset contains 95,502 insecure cases (4,521 cases with *state changes after external calls*, 80,825 cases with *missing checks for out-of-bound arithmetic operations*, and 10,156 cases with *missing checks for failing external calls*) in 28,621 contracts (as shown in Table II).

Environment. All experiments were carried out on a server running 64-bit Ubuntu 18.04 with two Intel Xeon Gold 5122 processors (8 cores each at 3.60GHz) and 128GB RAM.

B. RQ1: Scalability

To answer **RQ1**, we evaluated SMARTSHIELD against all 28,621 insecure contracts in our dataset.

Semantic Extraction. Since the size of each smart contract is limited to 0x6000 bytes [32], we set a timeout threshold for semantic extraction as 30 minutes per contract. In total, SMARTSHIELD successfully analyzed 27,476 (96.0%) out of the 28,621 insecure contracts and extracted their bytecode-level semantic information. For the remaining 1,145 contracts, SMARTSHIELD failed to analyze them because of the following reasons:

- ***Irregular control flows:*** The analysis of 773 (2.7%) insecure contracts did not terminate within the timeout limitation (i.e., 30 minutes). We found that these contracts contain complicated and unstructured control flows [33] (e.g., abnormal selection path, overlapping loops, parallel loops), which are either generated by an older version compiler or intentionally implemented by developers. It is important to note that although some of these contracts may be parsed successfully if we set a longer timeout threshold, a timeout of 30 minutes is a reasonable time trade-off throughout our experimental duration.
- ***Irresolvable target addresses:*** The analysis of 286 (1.0%) insecure contracts terminated half-way as these contracts contain irresolvable target addresses. Mostly, the target address of instruction `JUMP` or `JUMPI` is pushed onto the stack in advance by the instruction `PUSH`. However, in very rare cases, the target address is computed depending on an input data, namely, calldata. SMARTSHIELD is unable to resolve the concrete value of such target address and thus fails to accomplish the analysis. It is worth to mention that we can infer the concrete values of these target addresses from historical transactions related to each contract. As SMARTSHIELD is designed to revise insecure code patterns existed in each contract **before** it is deployed to the blockchain, no historical transaction is available at that point. Therefore, in our experiments, we did not import any historical transaction data from the blockchain to address this issue.
- ***Malfunctions:*** 86 (0.3%) insecure contracts failed to be analyzed due to their malfunctions. These malfunctioning contracts perform illegal operations that cannot be executed by the EVM. For example, an instruction pops operands from an empty stack or pushes operands onto a full stack. Interestingly, we observed that all these contracts do not have any related historical transactions, which indicates that they have been abandoned once being deployed to the blockchain. SMARTSHIELD does not consider these malfunctioning contracts as protection targets.

Contract Rectification. Given 95,502 labeled insecure cases in the 28,621 contracts, SMARTSHIELD then fixed these insecure cases and generate secure versions of contract EVM bytecode. Table III presents the number of fixed insecure cases and the number of rectified contracts, which includes fully rectified and partially rectified ones. In our dataset, 87,346 (91.5%) insecure cases were fixed by SMARTSHIELD and 25,060 (87.6%) out of the 28,621 insecure contracts were fully rectified. Besides, SMARTSHIELD marked 8,156 cases of insecure code in the remaining 3,561 contracts as "unrectifiable" because some potential side effects may be triggered by the rectification. To guarantee that the rectification does not introduce any potential side effects, SMARTSHIELD adopted a conservative policy to report the unfixable insecure cases to the developer without conducting any rectification. Note that if an insecure contract contains both fixable and unfixable insecure cases, SMARTSHIELD first fixes those that are fixable and then sends the rest together with the partially rectified contract to the developer.

Efficiency. SMARTSHIELD averagely spent 39 seconds to analyze and rectify an insecure contract. For static code analysis, SMARTSHIELD cost 28 seconds on average to extract bytecode-level semantic information from each contract, and 89.2% contracts in our dataset were analyzed within 10 seconds. On average, SMARTSHIELD took 11 seconds to fix the insecure cases in each contract and generate the rectified contract.

C. RQ2: Correctness

To verify the correctness of the rectified contracts, we first evaluated whether SMARTSHIELD actually fixed the insecure code in contracts. We applied the most prevalent techniques for analyzing smart contract, i.e., symbolic execution and abstract interpretation, to examine each rectified contract.

In particular, we leveraged three state-of-the-art smart contract analysis tools from both academia and industry, *Securify* [6], *Osiris* [16], and *Mythril* [17] to conduct the analyses. These tools not only identify security bugs, re-entrancy bugs, arithmetic bugs, unchecked return value bugs, respectively, but also locate the corresponding problematic instructions. These bug information are adequate for us to determine the existence of the pre-defined three insecure code patterns. After examining the outputs of those smart contract security analysis tools, we confirmed that previously labeled insecure cases are fixed in all rectified contracts, and each rectification does not introduce any other insecure cases.

Second, we evaluated whether our rectified contracts could defend against existing attacks. To replay such attacks, we collected historical transactions that are related to each rectified contract and retrieved the initial blockchain states of these transactions via *Infura* [34], an Ethereum infrastructure website. In total, we retrieved 22,527,186 related historical transactions. Then, we leveraged the open source EVM instance included in the *Go Ethereum* [30] project to re-execute contracts. We actually found suspicious historical transactions (e.g., triggering the inserted DataGuards) targeting

TABLE III: RESULTS OF CONTRACT RECTIFICATION

Category	# of eliminated cases	# of uneliminable cases	# of rectified contracts	
			Fully	Partially
CP.1	3,567	954	573	153
CP.2	74,642	6,183	21,815	3,655
CP.3	9,137	1,019	4,362	449
Total	87,346	8,156	25,060*	3,561*

∗ Some contracts contain multiple insecure patterns.

CP.1: State Changes after External Calls
CP.2: Missing Checks for Out-of-Bound Arithmetic Ops
CP.3: Missing Checks for Failing External Calls

on 47 contracts. By manually inspecting the source code of these contracts and the corresponding suspicious transactions, we found that the execution of these transaction violate the original intention of the insecure contracts. Fortunately, our rectified contracts are not affected by those transactions any more.

TABLE IV: EXISTING HIGH-PROFILE ATTACKS

Insecure contract	Category	Date of attack
*DAO** [35], [36]	CP.1	Jun. 17th, 2016 [25]
LedgerChannel [37]	CP.1	Oct. 7th, 2018 [38]
BeautyChain [39]	CP.2	Apr. 22nd, 2018 [26]
SmartMesh [40]	CP.2	Apr. 24th, 2018 [41]
UselessEthereumToken [42]	CP.2	Apr. 27th, 2018 [43]
Social Chain [44]	CP.2	May 3rd, 2018 [45]
Hexagon [46]	CP.2	May. 18th, 2018 [47]
KotET [48]	CP.3	Feb. 6th, 2016 [49]

∗ The *DAO* and the *DarkDAO* contract are considered to be identical.
CP.1: State Changes after External Calls
CP.2: Missing Checks for Out-of-Bound Arithmetic Ops
CP.3: Missing Checks for Failing External Calls

In addition, we replayed exploits of existing high-profile attacks[4], as summarized in Table IV, against rectified versions of those victim contracts. We collected the corresponding historical attack transactions that exploited each insecure contract and replayed them to attack its rectified version. Consequently, all the rectified contracts were protected against the malicious transactions. It demonstrates that SMARTSHIELD can protect smart contracts against the real-world threats effectively.

Finally, we validate whether the functionalities of each rectified contract are still executed consistently. To conduct the validation, we again used historical transaction data to re-execute each rectified contract and checked whether the implemented functionalities are executed still as the same. Theoretically, the *Ethereum* blockchain can be deemed as a transaction-based state machine. Contracts are triggered and executed by transactions, and the blockchain states are updated accordingly. Hence, we locally replayed the historical transactions and monitored final blockchain states. We executed each transaction twice, separately on the original and the rectified contract based on the same initial blockchain state. Then we compared the final blockchain states of these two executions. If the final states of the transaction executed on both the rectified and the insecure contract are the same, the rectification will be marked as "passed". Otherwise, an inconsistency is reported.

Note that some popular contracts are related to a large number of transactions (e.g., up to 3,078,930 transactions were sent to a single contract). It is inefficient and unnecessary to replay all these transactions. Therefore, under the premise that a high code coverage is achieved, we chose a proper number of transactions to replay for these contracts. We analyzed 100,000 transactions and observed that more than 70% code of most contracts can be executed by replaying 50 historical transactions. To balance the validation effectiveness and code coverage, we randomly replayed 50 historical transactions for each contract even if it involved more than 50 transactions.

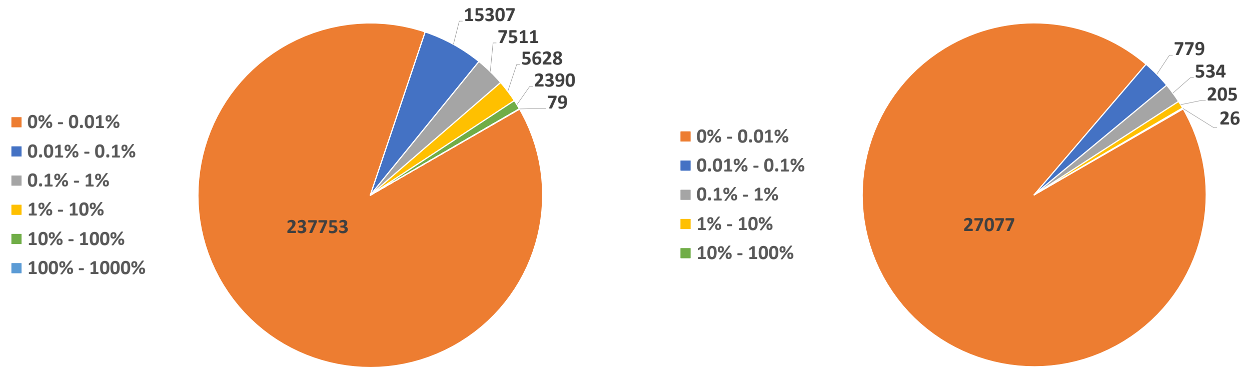
In our evaluation, SMARTSHIELD replayed 268,939 historical transactions. SMARTSHIELD identified that only 271 transactions produced divergent final blockchain states, in which 60 out of the 28,621 rectified contracts are involved. Except the malicious transactions that affected 47 contracts and therefore caused divergent results, we found that the other inconsistencies are due to an incompatible issue. The EVM bytecode of 13 contracts are incompatible with the latest EVM implementation, and thus cannot pass our replay. We checked these contracts and discovered that they were created before 2017 and have been abandoned for a long time. As the *Ethereum Mainnet* has evolved significantly through hard forks for upgrading, numerous improvements have been made to the EVM (e.g., gas cost, exception handling). As a result, these obsolete contracts are no longer supported by the latest EVM implementation. Since SMARTSHIELD is to protect a contract before its deployment, this incompatibility issue will not affect our rectification in practice.

D. RQ3: Cost

To answer **RQ3**, we counted the number of instructions inserted for each rectification and compared the gas consumption before and after contract rectification. For each rectified contract, SMARTSHIELD inserted 43.6 instructions on average, and the average size increment of each contract is around 1.0% (i.e., 49.3 bytes). SMARTSHIELD inserted an average of 9.7 instructions to fix each insecure case.

We further compared the gas consumption of each rectified contract and its corresponding insecure contract. We computed the gas consumption of the 268,668 historical transactions successfully replayed. Figure 7 depicts the increment of gas consumption after our rectification. On average, the gas con-

[4]For instance, the infamous DAO attack [25] exploited the *state changes after external calls* pattern in the *DAO* contract [35] and the *Dark DAO* contract [36] on June 17th, 2016.



(a) Consumption increment for replayed transactions

(b) Consumption increment for rectified contracts

Fig. 7: The increment of gas consumption after rectification

sumption for each rectified contract increases by 0.2%, that is, 64.0 extra units of gas (i.e., 0.00013 USD[5]) is required. For the replayed historical transactions, the average gas consumption increment for each transaction is around 35.4 extra units (0.3%) of gas (0.00007 USD). More specifically, the increments for 94.4% transactions (i.e., 253,623 out of the 268,668 transactions) are less than 0.1%. This clearly demonstrates SMARTSHIELD rectifies insecure contracts economically because the gas consumption increments are negligible.

E. Manual Validation

To gain an in-depth insights against rectification results, we randomly selected 200 insecure contracts to build the ground truth and manually inspected their rectifications. Among these contracts, there exist 126 cases of *state changes after external calls*, 278 cases of *missing checks for out-of-bound arithmetic operations*, and 188 cases of *missing checks for failing external calls*. We asked a team of annotators (two research fellows and two PhD students), all with more than three years of research experience in code repair area, to check whether the rectified contracts are correct. First, we asked the team members to check the rectified contracts independently. Then, all the members went through the results together and discussed those that are marked differently to reach an agreement. After the review, the team members confirmed that all the 592 insecure code pattern cases in those contracts are fixed effectively. In addition, they marked 194 contracts as "correctly rectified" and the other six contracts as "controversial".

For the six controversial cases, four of the insecure cases are recognized to be intentionally introduced by the developer to fulfill certain functionalities. For instance, a contract uses an *uint8* integer variable to implement a counter counting from 0 to 255. The developer intentionally omitted the overflow checks for counter increment and reset the counter to zero after reaching 256 through an overflow. As SMARTSHIELD is unable to accurately infer the intention of the developer through code analysis, the priority of SMARTSHIELD is to reduce the security risk of smart contracts.

[5]This price is calculated by the average gas price and the *Ethereum* (ETH) price on January 2nd, 2019.

```
1  mapping (address => uint) public balances;
2  bool private lockBalances;
3  ...
4  function withdraw(uint amount) public {
5    require(!lockBalances && balances[msg.sender] >= amount);
6    lockBalances = true;
7    if (msg.sender.call(amount)()) {
8      ...
9    }
10   lockBalances = false;
11 }
```

Fig. 8: A controversial case of rectified contract

The remaining two controversial cases are labeled by the analysis tools as vulnerable to re-entrancy attacks. However, the developer implemented an alternative approach to protect these contracts. The contract in Figure 8 is given as an example. The *withdraw* function updates the state variable *lockBalances* at line 10 after calling a function in an external contract at line 7. The execution sequence of the state change and the external call match the pre-defined insecure code pattern – *state changes after external calls*. However, the state variable *lockBalances* is a mutex that is introduced by the developer to protect the contract against re-entrancy attacks. The mutex is claimed at line 6 before the external call at line 7 and then released at line 10 after the call. SMARTSHIELD moves the release at line 10 to the front of the external call (line 7), which makes the mutex useless. Actually, our reviewer agreed that the mutex used in this contract is not optimal and may lead to new issues. For instance, the mutex may cause deadlocks. Moreover, because of the operations of storage read and write, the use of gas may also be significantly increased when the mutex is claimed and released. In comparison, SMARTSHIELD is designed to follow the best practice of checks-effects-interactions [27], which makes the rectification more concise and economical. When dealing with such controversial cases, the rectification report generated by

SMARTSHIELD is valuable as a reference for the developer to improve their code.

V. Related Work

A. Bug Analysis in Smart Contracts

Smart contract are vulnerable to severe security bugs. In many cases, these bugs are caused because of the fundamental differences between execution environments of smart contracts and traditional programs. Hence, it is difficult to develop smart contracts correctly, even for a simple one [50]–[53]. To detect bugs in smart contracts, numerous approaches and tools have been proposed recently. Based on their underlying techniques, previous works are classified into four major categories: symbolic execution, abstract interpretation, formal verification, and run-time monitoring.

Symbolic Execution. *Oyente* [2] detects timestamp dependence, transaction-ordering dependence, mishandled exceptions, and re-entrancy vulnerability in smart contracts by symbolically analyzing their EVM bytecode. Similarly, *teEther* [3] symbolically infers a sequence of state changing transactions and a final critical transaction to create exploits for vulnerable contracts, and *Maian* [54] identifies smart contracts that either lock funds indefinitely, leak them carelessly to arbitrary users, or can be killed by anyone. Since symbolic execution only explores restricted execution paths, it is unsound to infer semantic information declared in smart contracts. Instead, SMARTSHIELD adopts an abstract execution to extract semantic information as completely as possible.

Abstract Interpretation. *Securify* [6] uses the defined compliance and violation patterns to describe the given security properties. It then recognizes the conformance and non-conformance to practical security properties in smart contracts. *Zeus* [7] translates the source code into LLVM bitcode by using a formal abstraction of *Solidity* execution semantics and further verifies the correctness and fairness policies in smart contracts.

Formal Verification. Bhargavan *et al.* [55] analyzed and verified both the run-time safety and the functional correctness of smart contracts by translating the *Solidity* source code into F*. Grishchenko *et al.* [56] formally defined several crucial security properties for smart contracts in F* and checked these properties over smart contracts. Hirai *et al.* [57] proved some safety properties of smart contracts in an interactive theorem prover, Isabelle/HOL. Amani *et al.* [58] then extended the formalization and defined a sound program logic to verify smart contract at the bytecode level.

Run-Time Monitoring. *Sereum* [59] protects smart contracts against re-entrancy attacks by monitoring their executions. It performs taint tracking and introduces locks to state variables during smart contract executions. Besides, another smart contract run-time monitor, *ECFChecker* [60], checks whether a transaction violates a pre-defined general safety property, effectively callback free (ECF). Both *Sereum* and *ECFChecker* require modifications to the EVM, which causes extra execution overhead. Hence, these systems with modified EVM are hard to be adopted by the *Ethereum* community in reality.

In Addition, *ContractFuzzer* [8] proposes a set of test oracles and generates fuzzing inputs to test smart contracts. Numerous EVM bytecode decompilers (e.g., *Erays* [61], *Gigahorse* [62], *Porosity* [63], *Vandal* [64]) are also available to support manual bug analyses on smart contracts.

B. Bug Repair

Given the identified bugs in smart contracts, we discuss whether the existing patch approaches can be applied to rectify vulnerable contracts. Based on the target code, bug patch approaches are classified into two types: source-based [19], [23], [24], [65], [66] and bytecode-based [21], [22], [67].

Source-Based Analysis. *BovInspector* [66] performs rule-based patch to proceed buffer overflow in C programs. Based on the predefined rules (i.e., adding boundary checks and replacing unsafe APIs), *BovInspector* completes the path conditions and the buffer overflow constraints in programs. To patch bugs more effectively, *Vurle* [19] uses machine learning algorithm to generates templates.

Bytecode-Based Analysis. Without accessing source code, *AppSealer* [21] tracks the propagation of sensitive information and inserts shadow statements to taint and block dangerous data flows. Similarly, *CDRep* [67] patches cryptographic misuses based on the pre-defined templates. Both *AppSealer* and *CDRep* are designed for Android apps. However, smart contracts is far different from Android apps at the bytecode level because EVM bytecode does not have the notion of structs or objects, nor does it have a concept of methods [68]. Various relevant syntax and semantic information should be considered simultaneously when patching a smart contract.

VI. Conclusion

We proposed SMARTSHIELD, an automatic bytecode rectification system to fix insecure cases with insecure code patterns in smart contracts. SMARTSHIELD extracts bytecode-level semantic information and utilizes them to transform insecure contracts into secure ones. Through using SMARTSHIELD to handle 95,502 insecure cases in 28,621 real-world buggy smart contracts with source code, We demonstrated that SMARTSHIELD effectively fixed insecure cases and generates secure bytecode.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback. This work was partially supported by the Key Program of National Natural Science Foundation of China (Grant No.U1636217), the General Program of National Natural Science Foundation of China (Grant No.61872237), and the National Key Research and Development Program of China (Grant No.2016QY071401). We especially thank Ant Financial Services Group for the support of this research within the *SJTU-AntFinancial Security Research Centre*.

REFERENCES

[1] "A major vulnerability has frozen hundreds of millions of dollars of ethereum," https://techcrunch.com/2017/11/07/a-major-vulnerability-has-frozen-hundreds-of-millions-of-dollars-of-ethereum/.

[2] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 254–269.

[3] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1317–1333.

[4] C. F. Torres and M. Steichen, "The art of the scam: Demystifying honeypots in ethereum smart contracts," *arXiv preprint arXiv:1902.06976*, 2019.

[5] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 531–548.

[6] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 67–82.

[7] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *25th Annual Network and Distributed System Security Symposium, NDSS*, 2018, pp. 18–21.

[8] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 259–269.

[9] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 363–373.

[10] "Security considerations," https://solidity.readthedocs.io/en/latest/security-considerations.html.

[11] "Ethereum smart contract security best practices," https://consensys.github.io/smart-contract-best-practices/.

[12] "Solidity security: Comprehensive list of known attack vectors and common anti-patterns," https://github.com/sigp/solidity-security-blog.

[13] "Decentralized application security project (or dasp) top 10 of 2018," https://www.dasp.co/.

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

[15] "25% of all smart contracts contain critical bugs," https://news.bitcoin.com/25-of-all-smart-contracts-contain-critical-bugs/.

[16] C. F. Torres, J. Schütte *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 664–676.

[17] "Mythril," https://github.com/ConsenSys/mythril.

[18] "Watch out for insecure stackoverflow answers!" https://www.attackflow.com/Blog/StackOverflow.

[19] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "Vurle: Automatic vulnerability detection and repair by learning from examples," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 229–246.

[20] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 17, 2018.

[21] M. Zhang and H. Yin, "Appsealer: automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications." in *NDSS*, 2014.

[22] M. T. Azim, I. Neamtiu, and L. M. Marvel, "Towards self-healing smartphone software via automated patching," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 623–628.

[23] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "Autopag: towards automated software patch generation with source code root cause identification and repair," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 2007, pp. 329–340.

[24] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[25] "Understanding the dao attack," http://www.coindesk.com/understanding-dao-hack-journalists/.

[26] "Alert: New batchoverflow bug in multiple erc20 smart contracts (cve-2018-10299)," https://blog.peckshield.com/2018/04/22/batchOverflow/.

[27] "Use the checks-effects-interactions pattern," https://solidity.readthedocs.io/en/latest/security-considerations.html\#use-the-checks-effects-interactions-pattern.

[28] "Solidity," https://solidity.readthedocs.io/en/latest/.

[29] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[30] "Go ethereum," https://github.com/ethereum/go-ethereum.

[31] "Ethereum (eth) blockchain explorer," https://etherscan.io/.

[32] "Contract code size limit," https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md.

[33] M. H. Williams, "Generating structured flow diagrams: the nature of unstructuredness," *The Computer Journal*, vol. 20, no. 1, pp. 45–50, 1977.

[34] "Scalable blockchain infrastructure," https://infura.io/.

[35] "The dao contract," https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413.

[36] "The darkdao contract," https://etherscan.io/address/0x304a554a310C7e546dfe434669C62820b7D83490.

[37] "The ledgerchannel contract," https://etherscan.io/address/0xf91546835f756da0c10cfa0cda95b15577b84aa7.

[38] "Spankchain loses $40k in hack due to smart contract bug," https://www.coindesk.com/spankchain-loses-40k-in-hack-due-to-smart-contract-bug.

[39] "The beautychain (bec) contract," https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d.

[40] "The smartmesh (smt) contract," https://etherscan.io/address/0x55f93985431fc9304077687a35a1ba103dc1e081.

[41] "New proxyoverflow bug in multiple erc20 smart contracts (cve-2018-10376)," https://blog.peckshield.com/2018/04/25/proxyOverflow/.

[42] "The uselessethereumtoken (uet) contract," https://etherscan.io/address/0x27f706edde3ad952ef647dd67e24e38cd0803dd6.

[43] "Your tokens are mine: A suspicious scam token in a top exchange," https://blog.peckshield.com/2018/04/28/transferFlaw/.

[44] "The social chain (sca) contract," https://etherscan.io/address/0xb75a5e36cc668bc8fe468e8f272cd4a0fd0fd773.

[45] "New multioverflow bug identified in multiple erc20 smart contracts (cve-2018-10706)," https://blog.peckshield.com/2018/05/10/multiOverflow/.

[46] "The hexagon (hxg) contract," https://etherscan.io/address/0xb5335e24d0ab29c190ab8c2b459238da1153ceba.

[47] "New burnoverflow bug identified in multiple erc20 smart contracts (cve-2018-11239)," https://blog.peckshield.com/2018/05/18/burnOverflow/.

[48] "The kotet contract," https://etherscan.io/address/0xb336a86e2feb1e87a328fcb7dd4d04de3df254d0.

[49] "Post-mortem investigation (feb 2016)," https://www.kingoftheether.com/postmortem.html.

[50] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "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, 2016, pp. 79–94.

[51] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[52] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia, "Dissecting ponzi schemes on ethereum: identification, analysis, and impact," *Future Generation Computer Systems*, vol. 102, pp. 259–277, 2020.

[53] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 478–493.

[54] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 653–663.

[55] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2016, pp. 91–96.

[56] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.

[57] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 520–535.

[58] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 2018, pp. 66–77.

[59] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[60] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 48, 2017.

[61] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: reverse engineering ethereum's opaque smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1371–1385.

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

[63] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *DEF CON*, vol. 25, p. 11, 2017.

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

[65] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in *European Symposium on Research in Computer Security*. Springer, 2010, pp. 71–86.

[66] F. Gao, L. Wang, and X. Li, "Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 786–791.

[67] S. Ma, D. Lo, T. Li, and R. H. Deng, "Cdrep: Automatic repair of cryptographic misuses in android applications," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 711–722.

[68] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 116, 2018.