Gap between Theory and Practice : An Empirical Study of Security Patches in Solidity

Anonymous Author(s)

ABSTRACT

Ethereum, one of the most popular blockchain platforms, provides financial transactions like payments and auctions through smart contracts. Due to the tremendous interest in smart contracts in academia, the research community of smart contract security has made a significant improvement recently. Researchers have reported various security vulnerabilities in smart contracts, and developed static analysis tools and verification frameworks to detect them. However, it is unclear whether such great efforts from academia has indeed enhanced the security of smart contracts in reality.

In this paper, we empirically studied how secure real-world smart contracts are in the case of the Solidity programming language. We first examined how many well-known vulnerabilities the Solidity compiler has patched, and how frequently the Solidity team publishes compiler releases. Unfortunately, we observed that many known vulnerabilities are not yet patched, and some patches are not even sufficient to avoid their target vulnerabilities. Subsequently, we investigated whether smart contract developers use the most recent compiler with vulnerabilities patched. We reported that developers of more than 98% of real-world Solidity contracts still use older compilers without vulnerability patches, and more than 25% of the contracts are potentially vulnerable due to the missing security patches. To understand actual impacts of the missing patches, we manually investigated potentially vulnerable contracts, and identified common mistakes by Solidity developers, which may cause serious security issues such as financial loss. We reported hundreds of vulnerable contracts: three have been assigned CVE IDs, and more are requested. About one fourth of the vulnerable contracts are used by thousands of people. We recommend the Solidity team to make patches that resolve known vulnerabilities correctly, and developers to use the latest Solidity compiler to avoid missing security patches.

KEYWORDS

Empirical Study, Smart Contracts, Solidity, Security Patches

ACM Reference Format:
Anonymous Author(s). 2019. Gap between Theory and Practice : An Empirical Study of Security Patches in Solidity. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Blockchain technologies have become increasingly popular and are now considered as one of the key enablers of secure distributed computations providing, a wide spectrum of applications with *smart contracts*. One of the most popular blockchain platforms is Ethereum [3, 5], which implements a decentralized Turing-complete virtual machine, called the Ethereum Virtual Machine (EVM), and it can support not only cryptocurrency but also other applications like games and financial services [4]. The key to support such a variety of applications in the blockchain is smart contracts.

Smart contracts are programs that provide security-critical applications like financial transactions, but they are vulnerable to various security attacks [6]. Among languages that are compiled to Ethereum bytecode, embedded in transactions as bytecode, and run on EVM, Solidity [45] is the most widely supported and maintained one so far. However, because the semantics of Solidity is not formally nor completely specified, severe security issues with smart contracts have been reported. For example, vulnerabilities of the Decentralized Autonomous Organization (DAO) [2] resulted in $150 million being stolen, and Parity's $280M Ethereum wallets [1] had to be frozen.

To alleviate the problem of insecure smart contracts, researchers have studied various security vulnerabilities, and developed static analysis tools and verification frameworks to detect them. Static analysis techniques to detect such vulnerabilities formally verify a subset of Solidity [8], analyzes Ethereum bytecode using symbolic execution [24, 25, 27, 29], or analyzes Solidity contracts by compiling them to LLVM bitcode [23]. Researchers also have attempted to formalize the semantics of various subsets of Solidity [18, 20, 21, 28, 34]. Such efforts from academia made a significant improvement in the theory of smart contract security.

However, it is unknown whether such efforts from academia improved the security of smart contracts in reality. To the best of our knowledge, no previous research studied whether the Solidity language becomes more secure by fixing known vulnerabilities. Does the Solidity compiler get security patches for the known vulnerabilities? If so, would Solidity developers take advantage of the repaired Solidity compilers? Are the compiler patches strong enough to prevent their target vulnerabilities? In addition, do developers actually make bugs and errors due to such vulnerabilities?

In this paper, we report our empirical findings about the gap between the important advances in the theory of smart contract security and the development and uses in real-world contracts. After collecting 41 known vulnerabilities from diverse sources including academic papers and industrial reports, we examined all the Solidity patches from the official release notes [13] and found that seven out of 41 known vulnerabilities are patched, and one patch is under development by the Solidity compiler team. Thus, we observed that many known vulnerabilities are not yet patched, and even some patches applied to prevent specific vulnerabilities are not sufficient to remove the them.

Given that the Solidity team has patched at least seven known vulnerabilities, would Solidity developers utilize the Solidity compiler fixes? To understand whether Solidity developers use security patches in their contract development, we investigated whether the developers use the most recent compiler version with vulnerabilities patched. Unfortunately, we found out that developers of 98.14% of Solidity contracts in the wild used older compilers than the up-to-date version at the time of developing the contracts. Among 55,046 contracts that are currently available from Etherscan [16], 54,021 contracts are created by the compilers without all the security patches available. To see the effects of not using the most recent compiler versions, we built a light-weight static analyzer that detects contracts exposed to unpatched vulnerabilities. The analyzer reported that 13,943 contracts (25.33%) are potentially vulnerable due to the missing security patches.

In order to understand whether the potentially vulnerable contracts reported by our tool actually contain security issues, we manually investigated most popular contracts among reported ones. For each vulnerability, we inspected how the Solidity compiler patched the vulnerability, whether the patch is sufficient, what mistakes developers commonly make due to the vulnerability, and what kinds of security issues they can cause. As far as we know, previous research did not conduct in-depth investigation on most of the vulnerabilities discussed in this paper such as *storage variable shadowing confusion* and *inheritance order confusion*. Our investigation showed that hundreds of vulnerable contracts are doubtlessly exploitable, received three CVE IDs, and requested more CVE IDs. Furthermore, our work also revealed insecure coding practices from Solidity developers that have not been previously studied.

We recommend the Solidity team to make security patches that resolve known vulnerabilities correctly, and developers to use the latest Solidity compiler to avoid missing security patches. All the datasets and tool used for our empirical study are publicly available[1]. In the remainder of this paper, we discuss the findings in detail.

2 BACKGROUND

Before discussing the findings, this section explains basic features of Solidity to understand the paper.

2.1 Solidity Basics

Solidity [45] is a high-level programming language developed by Ethereum Foundation. While Ethereum transactions embed smart contracts in the form of EVM bytecode, programmers often write contracts in high-level languages, which is less error-prone and more productive than in bytecode. Solidity is the most supported and widely-used language that compiles to EVM bytecode.

A smart contract source unit is a sequence of contract definitions. A contract definition defines a contract, a library, or an interface. A contract is like a class in object-oriented languages that support multiple inheritances. Contracts can specify their fields and methods. A contract resides at a specific address on the Ethereum blockchain. The EVM is the runtime environment for Solidity contracts, which is sandboxed and isolated. Solidity uses five memory areas to save data of contracts: storage, stack, memory, logs, and calldata. Among them, storage is the only persistent memory area, and the other four are temporal ones.

[1]Anonymized due to a double-blind review process

2.2 Storage

Every contract has a persistent key-value store, which maps 256-bit words to 256-bit words, called *storage*. Each key refers to a "slot" and each slot can store 256-bit data; thus, storage is a collection of slots. A contract can neither read nor write to any other storage than its own.

The storage layout is defined and fixed at the creation time of a contract, and it cannot be altered at run time [45]:

> Statically-sized variables (everything except mapping and dynamically-sized array types) are laid out contiguously in storage starting from position 0. Multiple items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules: ⋯ Structs and array data always start a new slot and occupy whole slots ⋯

Thus, each element of a fixed-size `uint` array takes the entire single 256-bit slot in storage. Because the sizes of mapping types and dynamic array types are unknown at compile-time, such types use a Keccak-256 hash function to calculate the starting point of the value or the array data.

3 SOLIDITY SECURITY PATCHES

First, we explored the question of whether the Solidity programming language becomes more secure by fixing known vulnerabilities. We examined the question in two respects: how many known vulnerabilities are patched by the Solidity compiler and how often the compiler gets released.

3.1 Does Solidity Patch Known Vulnerabilities?

We first collected known vulnerabilities from various sources including lists from industry and a thorough collection from academic papers as follows:

(1) The most recent version of the official Solidity documentation as of March 31, 2019 [45]
(2) SWC registry: Smart Contract Weakness Classification and Test Cases [47]
(3) ConsenSys: Ethereum Smart Contract Best Practices [44]
(4) NCC Group: Decentralized Application Security Project [19]
(5) 186 academic papers from major conferences and arXiv that contain "smart contract" in their titles and abstracts

The SWC registry is an implementation of the weakness classification scheme proposed in EIP-1470 [12]; its purpose is to share security issues in smart contracts. The ConsenSys group assembled "Ethereum Smart Contract Best Practices" and the NCC Group runs the "Decentralized Application Security Project", which is a collaborative effort discover smart contract vulnerabilities within the security community. We inspected all the sources and collected all the reported vulnerabilities from them, which amounts to 41. The list of collected known vulnerabilities is publicly available[2].

[2]Anonymized due to a double-blind review process

Table 1: Known vulnerabilities patched by the Solidity compiler and the number of contracts that are vulnerable to them

ID	Known Vulnerability	Patched Version	No. of Vul. Contracts	No. of Vul. Contracts with ≥ 10 tx	Total No. of tx of Vul. Contracts with ≥ 10 tx
SPV-01	Misuse of constructors	0.5.0	482	109	359,671
SPV-02	Functions without visibility	0.5.0	754	363	3,604,633
SPV-03	Storage variable shadowing confusion	Planned	493	231	895,828
SPV-04	Type casting to arbitrary contracts	0.4.0 and 0.5.0	864	374	1,940,529
SPV-05	Inheritance order confusion	0.4.24 and 0.5.0	475	177	1,898,334
SPV-06	Uninitialized storage pointers	0.5.0	89	32	159,975
SPV-07	Typo of the += operator	0.5.0	4	4	106
SPV-08	Use of deprecated functions	0.5.0	10,782	3,342	22,164,034
Total			**13,943**	**4,632**	**31,023,110**

For each known vulnerability, we investigated whether any version of the Solidity compiler patched the vulnerability. We reviewed the official Solidity release notes [13] and double-checked them by implementing a contract with the vulnerability, compiling it with a patched and an unpatched versions of the Solidity compiler, and comparing their results. We found that seven out of 41 known vulnerabilities are patched and one is planned to be patched as summarized in Table 1. We named each vulnerability in the table with the "SPV" prefix, which stands for *Smart contract Patched Vulnerability*. The second column describes the vulnerability and the third column shows which Solidity compiler version patched the vulnerability. Note that a patch for SPV-03 is under development [46]. We discuss the remaining columns of the table in Section 5.

Because all the existing Solidity patches for known vulnerabilities are available from Solidity 0.5.0, we strongly recommend developers to use at least version 0.5.0 of the Solidity compiler.

3.2 How Often Does Solidity Get Updated?

In order to understand whether the Solidity team actively responds to demands from developers such as feature requests and security patches, we inspected the official Solidity release notes. From Solidity 0.1.2 released on August 21, 2015, 50 compiler releases have made till version 0.5.7 released on March 26, 2019. The maximum number of days between consecutive releases are 121, the minimum one, and the average days 27. Based on the data, we regard that the Solidity compiler is actively evolving and getting security patches as long as they are available.

4 USES OF SOLIDITY SECURITY PATCHES

Now that the Solidity compiler patched seven out of 41 known vulnerabilities, do developers use the compiler with patches? We studied the question in three regards: 1) how many contracts are compiled by older versions of the compilers than the up-to-date versions at the time of developing them, 2) how many contracts are exposed to patched vulnerabilities due to the use of old compilers, and 3) how many contracts are statically reported as vulnerable due to the use of old compilers.

To answer these questions, we developed a crawler to collect Solidity smart contracts from Etherscan [16], a well-known Ethereum blockchain explorer. For each contract, we collected its name, source code, compiled version, deployment date, Ether balance, number of transactions, and the last transaction date. We crawled them from December 14, 2018 to March 31, 2019, and collected 55,820 verified contracts. Among 55,820 contracts, we excluded 774 destructed contracts, which are unavailable for use. For the remaining 55,046 *live* contracts, we used their total numbers of transactions and the dates of the last transactions to identify the most actively used ones.

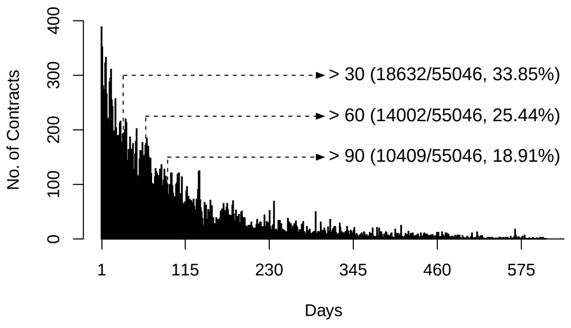

Figure 1: Available days for developers to make their contracts conform to newly released compiler versions

4.1 How Many Contracts Are Compiled by Old Compilers?

To get the number of contracts that are compiled by older versions of compilers, we checked the deployed dates of contracts and their used compiler versions. When the used compiler version of a contract is not the latest compiler version at the time of its deployment, we confirmed that the contract was compiled by an old compiler. Among 55,046 live contracts, 26,373 contracts were compiled by old compilers, which amounts to 47.91%. The result indicates that many developers do not use the latest version of the Solidity compiler.

To understand why the developers did not use the latest version of the Solidity compiler, we further investigated how many days are passed between the release dates of the newest compilers and the contract deployment dates. They show the numbers of available days for developers to make their contracts applicable to newly released versions of the compiler before deployment. Figure 1 illustrates that 33.85% of the contracts had more than 30 days, 25.44% had more than 60 days, and 18.91% had more than 90 days to use newly released compilers before deploying their contracts. The result shows that the developers of about one third of the contracts

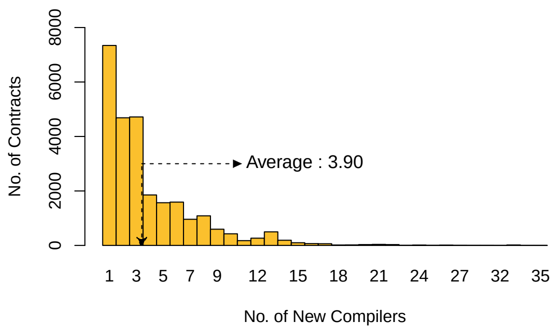

Figure 2: Number of compiler versions missed by developers

Table 2: Number of contracts that are not exposed to known vulnerabilities due to the use of old compilers

Compiled Version	No. of Contracts	Patched Vulnerabilities
0.5.0 $\leq$	1,025 (1.86%)	SPV-01 to SPV-08 except SPV-03
0.4.24 $\leq v <$ 0.5.0	21,638 (39.31%)	SPV-05 partially
$<$ 0.4.24	32,383 (58.83%)	None
Total	**55,046 (100%)**	

had more than 30 days to make their contracts comply to the latest compiler version.

In addition, for the contracts compiled by old compilers, we also checked how many compiler versions were missed between the newest ones and the used ones by developers. Figure 2 shows that most developers used the previous versions of the latest ones, and four compiler versions were missed on average.

4.2 How Many Contracts Are Exposed to Vulnerabilities?

Given that 47.91% of the contracts are compiled by old compilers, how many contracts are exposed to already patched vulnerabilities due to the use of old compilers? Table 2 summarizes the result[3]. More than a half of the live contracts, 58.83% were compiled by compilers prior to v0.4.24, and 98.14% were compiled by the ones below v0.5.0. Therefore, more than 98% of the live contracts are exposed to the vulnerabilities that are already patched in v0.5.0.

4.3 How Many Contracts Are Detected as Vulnerable?

Among 98% of the live contracts that are open to the vulnerabilities, how many of them are actually vulnerable? To understand the impacts of missing Solidity patches in practice, we developed a simple static analysis tool that detects potentially vulnerable contracts due to the lack of Solidity patches. We describe how the tool detects vulnerable contracts for each vulnerability in the next section.

Table 1 presents the results of our static analyzer. Out of 55,046 contracts, it reported that 13,943 contracts may be vulnerable due to the missing patches, which could have been prevented if the patches were applied. The contracts reported as vulnerable are popular ones as the numbers of their transactions, denoted as tx, show. Even such popular contracts with many transactions may have vulnerabilities that were already patched in the current compiler.

[3]We considered v0.4.26 as v0.5.0 because it was officially released after v0.5.0.

5 VULNERABLE CONTRACTS IN PRACTICE

To verify actual impacts of missing Solidity patches in reality, we manually investigated the potentially vulnerable contracts reported by our tool. To analyze only meaningful contracts, we chose contracts with at least 10 transactions. For each vulnerability, when over 300 contracts have more than or equal to 10 transactions, we selected top 200 contracts in the order of the number of total transactions.

In this section, we explain each vulnerability, how the Solidity compiler patches it, whether the patch is sufficient to avoid the vulnerability, how our tool detects possibly vulnerable contracts, what kinds of errors and mistakes developers actually make due to the vulnerability, and how the developers' mistakes can cause serious security problems. Code examples in this section are all from real-world contracts.

5.1 Misuse of Constructors

Solidity initially provide two ways to define contract constructors, but one has been deprecated from v0.5.0 because it often allows developers to make vulnerable contracts. The original two ways to make a function constructor are 1) using the `constructor` keyword and 2) using the same name as its contract's name. The second way is vulnerable because if a developer accidentally uses a different name for a function intended to be a constructor from its contract's name, the function becomes an ordinary function, which may be accessible by external users. Since constructors usually perform critical operations like initialization of sensitive data such as contract owners, if they are open to external attackers, it may cause security issues. Thus, the Solidity compiler from v0.5.0 forbids the second way by rejecting functions that have the same name with their contracts. Unfortunately, this patch is not sufficient to avoid the vulnerability, because programmers can still make mistakes like using different names for a function intended to be a constructor and its contract. As we show in this section, Solidity developers often make such mistakes in real-world smart contracts, and we received three CVE IDs for them[4].

Our tool reports potentially vulnerable contracts due to misuse of constructor names by using a well-known string similarity algorithm, Levenshtein distance [26], for function names and contract names. Because the algorithm gives the same weight for insertion and delete of letters and case substitution, we modified the algorithm using the weight rules proposed by Wagner and Fischer [51] to give different weights for each operator. In addition, we added a swap operator to detect similar strings that are different only in the order of two letters. The tool also analyzes comments; if a comment just above a function definition contains the `constructor` keyword ignoring cases, and if the function's name is different from its contract's name, the tool reports it as a vulnerable constructor.

[4]https://github.com/smsecgroup/SM-VUL/tree/master/typo-vul-00
https://github.com/smsecgroup/SM-VUL/tree/master/typo-vul-01
https://github.com/smsecgroup/SM-VUL/tree/master/typo-vul-02

As Table 1 shows, the tool detected 482 contracts as vulnerable due to misuse of constructors. We manually investigated 109 contracts with at least 10 transactions, and we confirmed that all of them are indeed vulnerable constructor functions in the sense that either their comments say that they are meant to be constructors or the function bodies perform only initialization. We further identified *exploitable* contracts that are callable by external users and let them change sensitive data such as their contract owners. ***We found out that 55 contracts, which amounts to 50.46%, are exploitable due to vulnerable constructors. These contracts are popular ones because the total number of their transactions is 63,674, and 13 contracts have more than 100 token holders.***

We analyzed 55 exploitable contracts and classified their main causes as the following three types of developer mistakes:

Incorrect Reuse of Standard Contracts: We observed that developers incorrectly reuse standard contracts' constructors provided by Ethereum and other renowned blockchain companies. For example, we encountered developers incorrectly reuse standard contracts such as `tokenERC20` [15] by Ethereum and `SimpleToken` [31] by OpenZeppelin [30]. When developers reused it, they often copied the contract and changed only the contract name leaving the constructor name unchanged, which makes a vulnerable constructor. We found that 18 different contracts had the same constructor of `tokenERC20`.

Typos in Constructor Names: We observed that developers do make typos in constructor names due to wrong capitalization, swapping characters, and misspelling. For example, we encountered 24 contracts with simple typos where contract names and constructor names differ only in one letter, one number, the order of two characters, or capitalization. They are popular contracts in that the total number of their transactions were 54,662, and we received three CVE IDs for them.

Usage of `public` Functions: We found that developers use `public` functions as constructors. If a `public` function is used as a constructor, its contract should guarantee that only authorized users can call the constructor. However, we found that ten contracts had no authentication, which allows external attackers to call them and alter their sensitive data, and another case checked only its initialization flag, which may lead to the *race condition* attack [32] as attackers may be able to call the function before the owner of the contract.

In addition to the three patterns from exploitable contracts, we also observed another frequent pattern from 24 non-exploitable contracts: splitting initialization functions in multiple functions. Consider the following minimized code from a real-world contract:

```
contract BookERC20EthV1p1 {
  // Constructor.
  // Creator needs to call init() to finish setup.
  function BookERC20EthV1p1() { ... }
  function init(...) public { ... }
}
```

where initialization statements are distributed in `BookERC20EthV1p1` and `init`. As the comment says, the contract owner should manually call all of the necessary functions to properly finish the initialization process. We analyzed transactions of these 24 contracts to see whether the owners of the contracts called all the necessary functions properly right after their contract deployment. We found out that six out of 24 contracts failed to call all the necessary functions, which may miss initialization of important data such as token addresses and the initial token balances of their owners. Even though they may cause such problems, we classified them as non-exploitable because they all properly authenticated function callers.

Finally, we report three insecure cases due to vulnerable constructors observed from exploitable contracts. First, 19 constructors initialized their owners to callers. Consider the following example:

```
contract Owned {
  address public owner;
  function owned() public { owner = msg.sender; }
}
```

where the `owned` function was intended to be a constructor but it is a public function due to the typo in its name. Any attacker that calls `owned` becomes the owner of the contract, and the attacker can stealing all the Ether reserved in the contract, obtain tokens without paying a fee, and more. Second, we found 38 vulnerable constructors that transfer their tokens to their callers. For example:

```
contract BossCoin {
  function TokenERC20(...) public {
    totalSupply = 2000000000;
    // Give the creator all initial tokens
    balanceOf[msg.sender] = totalSupply;
  }}
```

any attacker can get the total tokens of `BossCoin` by simply calling `TokenERC20`. Third, we found that vulnerable constructors initialize flags for crowdsale and ICO status. By calling such constructors, attackers can change the flag of a finished ICO and buy its tokens.

5.2 Functions without Visibility

Solidity used to allow function declarations to omit their visibilities, but from v0.5.0 it throws a compilation error if a function visibility is not specified. In Solidity, a function may declare its visibility as `external`, `internal`, `public`, or `private`. Functions with the `public` or `external` visibility can be called by external users, whereas `private` and `internal` functions are hidden from them. Hence, if attackers attempt to attack contracts, the only possible entry points are `public` and `external` functions. However, before v0.5.0, when a function declaration does not specify its visibility, it was `public` by default, which is critically vulnerable. Thus, Solidity enforces functions to declare their visibilities. Note that Android had a similar history. Since the visibility of an Android content provider used to be `public` by default, it caused many security issues [22] and Google officially changed the default visibility to false, which corresponds to `private`.

Our tool reports functions as potentially vulnerable due to function visibility when they satisfy the following three conditions: 1) no visibility specified, 2) no modifier function checking pre-conditions, and 3) update of a "critical storage variable", a storage variable that is used by modifier functions, because modifiers often check the validity of security-sensitive data stored in storage variables.

Table 1 shows that the tool detected 754 contracts as vulnerable due to function visibility. We chose 200 contracts in the decreasing order of the number of total transactions, and manually investigated them to see whether they are indeed exploitable. We considered contracts as exploitable if 1) their functions reported as vulnerable do not check authorization for updating critical storage variables, which allows any attacker to update them, and 2) updates on critical storage variables disable the validity checking of the storage variables in modifier functions. Thus, we may have missed some exploitable contracts that do not meet these requirements but we do not find false positives. ***Out of 200 contracts, we confirmed that 63 contracts have vulnerable functions, which amounts to 31.50%. They are popular ones whose total number of transactions is 1,065,643, and 48 contracts have more than 100 token holders.***

We analyzed 63 exploitable contracts and observed three types of developer mistakes: (1) only initialization checking, (2) no checking, and (3) misuse of constructors, again! We found that most vulnerable functions checked only the initialization flags before updating critical storage variables, which as we discussed in the previous section, can cause the race condition attack. We also found that vulnerable functions often missed any authentication logic, which allows anyone to call such functions to change critical storage variables. The third pattern is wrongly named functions that were intended to be constructors. Consider the following example from a real-world contract:

```
contract Token {
  function Ownable() { owner = msg.sender; }
  function transfer(address _to ...) ... {...}
  modifier onlyOwner() {
    require(msg.sender == owner); _; }
  ...}
```

which shows that its developer merged a standard example contract `Token` [9] and an `Ownable` contract [52] into one contract, but failed to rename the constructor of `Ownable`, which became a general `public` function. Now, any attacker can call the `Ownable` function to change the owner of the contract.

Our investigation revealed that attackers can change the following critical storage variables:

(1) important addresses such as the owners of the contracts, crowdsale contract addresses, proxy token addresses, proxy asset addresses, and wallet addresses
(2) ICO dates like the start date of ICO, the end date of ICO, and the start date of pre-invest

By changing the above critical storage variable, attackers can perform various critical attacks. For example, attackers can steal all the Ether balances reserved in a contract once they change the owner of the contract.

5.3 Storage Variable Shadowing Confusion

In Solidity, when a parent contract and a child contract have storage variables with the same name, the storage variable in the parent is hidden from the child. As a result, with the same name, functions in the parent refer to the variable defined in the parent, and functions in the child refer to the variable defined in the child, which could be confusing to developers. The Solidity team is currently working on a patch for this vulnerability and aims to publish it in the next release [46].

Our tool identifies contracts as potentially vulnerable due to storage variable shadowing confusion when they satisfy the following three conditions: (1) parent and child contracts have storage variables with the same name, (2) one of the functions from the parent contract reads or writes the storage variable with the same name defined in the parent to show the effects of shadowing, and (3) the visibility of the storage variable in the parent is `public` or `internal` because only such storage variables are inherited.

Table 1 shows that the tool detected 493 contracts as vulnerable due to storage variable shadowing confusion. We manually inspected 231 contracts that have more than or equal to 10 transactions. ***Among 231 contracts, we found out that 106 contracts are exploitable, which amounts to 45.89%. The total number of transactions for 106 contracts is 292,048, and 26 contracts have more than 100 token holders.***

We analyzed 106 exploitable contracts and observed six patterns of developers' mistakes.

Initialization Conflict: The first pattern is to initialize only the storage variable defined in the child without initializing the storage variable defined in the parent, which causes a problem when a function in the parent uses the uninitialized storage variable in the parent. Consider the following example:

```
 1 contract StandardToken is ERC20, SafeMath {
 2   uint public totalSupply;
 3   function burn(uint256 _value)
 4     public returns (bool success)
 5     { totalSupply = safeSub(totalSupply,_value); }
 6 } //end of parent contract
 7 contract USDXCoin is Ownable, StandardToken {
 8   uint public totalSupply;
 9   function USDXCoin() public
10     { totalSupply= 10*(10**6)*(10**6); }
11   function mintToken(uint _value...)
12     { totalSupply = safeAdd(totalSupply, _value) }
13 }
```

Both the parent `StandardToken` contract and the child `USDXCoin` contract have the storage variable named `totalSupply`, but only the `totalSupply` variable in the child is initialized on line 10. Hence, an integer underflow exception occurs if the `burn` function in the parent on line 3 is called.

Function Conflict: The second pattern denotes such cases when a developer splits functionalities into both parent and child contracts. Let us consider the above example again. Addition to the `totalSupply` variable is done only in the child contract by the `mintToken` function on line 11 and subtraction from `totalSupply` is done only in the parent contract by the `burn` function on line 3. Thus, `burn` in the parent updates `totalSupply` defined in the parent, whereas `mintToken` in the child updates `totalSupply` defined in the child, which may lead to unexpected behaviors due to inconsistent use of `totalSupply`.

Write Conflict: The third pattern is when a developer updates only one of the storage variables as shown in the following example:

```
 1 contract Ownable {
 2   address public owner;
 3   function Ownable() { owner = msg.sender; }
 4   function transferOwnership(address newOwner)
 5     onlyOwner public { owner = newOwner; }
 6 }
 7 contract DadiToken is StandardToken, Ownable {
 8   address public owner;
 9   function DadiToken (...) { owner = msg.sender; }
10 }
```

The parent `Ownable` and the child `DadiToken` both have the storage variable named `owner`. Initially, both variables have the same value on lines 3 and 9. However, once the `transferOwnership` function in the parent gets called, it updates only the `owner` variable defined in the parent leaving `owner` defined in the child intact.

Missing Override: The fourth pattern is when a child contract overrides only some of its parent's functions that use the storage variable defined in both parent and child. Then, the overridden functions in the child update the storage variable defined in the child, whereas non-overridden functions update the one defined in the parent.

Super Call Conflict: This pattern represents cases when a child contract overrides all of its parent's functions that use the storage variable defined in both parent and child, and one of the overriding functions in the child calls an overridden function in the parent internally. Because the called function in the parent uses the storage variable in the parent, such super calls could be confusing.

Override Misunderstanding: The last pattern is due to the confusion between overriding and overloading. Consider the following example:

```
1  contract Crowdsale {
2    uint256 public rate;
3    function transferToken(address beneficiary,
4                           uint256 weiAmount) internal
5      { uint256 tokens = weiAmount.mul(rate); }
6  }
7  contract CappedCrowdsale is Crowdsale { ... }
8  contract EthealNormalSale is CappedCrowdsale ... {
9    uint256 public rate = 700;
10   /// @dev Overriding Crowdsale#transferToken.
11   function transferToken(address _beneficiary,
12                          uint256 _weiAmount,
13                          uint256 _time, ...)
14                         internal {}
15 }
```

While the comment on line 10 claims that `transferToken` on lines 11–14 overrides the function in the parent on lines 3–5, because the former has more than two parameters and the latter has only two parameters, they are overloaded rather than overridden. As a result, a different `rate` is used depending on which function is called.

Finally, we report the following list of eight security issues that we found from real-world contracts:

- **Confusion of the total amount of tokens:** The `totalSupply` storage variable holds the total amount of available tokens. Due to shadowing on such a variable, the calculated amount of available tokens was wrong.
- **Confusion of user balances:** Storage variables like `balances` and `balanceOf` hold token balances of their users. Due to shadowing on them, the calculated amount of user balances was wrong.
- **Confusion of owners:** The `owner` storage variable stores the address of the contract owner. Because of shadowing on `owner`, only one of the storage variables from a parent contract and a child contract got updated, which led to confusion on owners.
- **References of null contracts:** Most crowdsale contracts have the `token` storage variable, which holds the address of the ERC20 token contract implementation. However, because this variable was shadowed, the correct address of the token was not available.
- **Confusion of the total amount of commission fees:** Because storage variables like `rate` were shadowed, the calculated amount of commission fees was different depending on whether a parent function or a child function was called.
- **Confusion of the sale duration:** Storage variables such as `cap` and `goal` decide the duration of token sales. Because such variables were shadowed between parent and child contracts, the functions in the parent and the child had different durations.
- **Confusion of the mint duration:** Since `mintingFinished` was shadowed between parent and child contracts, the functions in the parent and the child had different minting duration.
- **Transfer of invalid tokens:** The `allowed` storage variable holds the allowed token amount to be transferred. Since this variable was shadowed, the transferred amount was different depending on whether a function from a parent or a child was called.

Note that we encountered a possibly malicious contract that uses variable shadowing confusion. Consider the following contract:

```
1  contract StandardToken is Token {
2    uint256 public totalFee;
3    function transfer(address to, ...) { ...
4      totalFee = safeAdd(totalFee, Fee); ...
5    } ... }
6  contract HawalaToken is StandardToken {
7    uint256 public totalFee;
8    function disperseRewards(address to,
9                             uint256 amount) { ...
10     if (totalFee > 0 && totalFee > amount) {
11       totalFee = safeSub(totalFee, amount);
12       balances[to] = safeAdd(balances[to], amount);
13     ... } ...  } }
```

Because both parent and child contracts define the storage variable `totalFee`, the parent adds fees to its `totalFee` on line 4, and the child subtracts some amounts from its `totalFee` on line 11 as we discussed for the **Function Conflict** pattern. This behavior is not only unintuitive but possibly malicious. While the `disperseRewards` function in `HawalaToken` distributes rewards only if `totalFee` is greater than 0, because `totalFee` in `disperseRewards` is always 0, users cannot receive rewards. This behavior from a real-world contract could be due to a developer's mistake or an intentional malice, but it clearly shows that attackers can utilize the storage variable shadowing vulnerability to implement malware that look benign.

5.4 Type Casting to Arbitrary Contracts

Solidity allows explicit and implicit type conversion [42], and the old Solidity compilers allowed developers to perform bad-casting:

(1) down-casting: conversion of a parent contract type to a child contract type
(2) unrelated-casting: conversion of a contract type to another contract type that is not its parent nor its child contract type
(3) non-contract casting: conversion of a contract type to a non-contract type

Problems with bad-casting are twofold: calling a function from a wrongly converted type may call an unintended function, or simply perform no operation without throwing any error, depending on the existence of functions with the same name. To alleviate the problems, the Solidity compiler from v0.4.0 throws an exception if an address associated with no code is called, and from v0.5.0 it disallows down-casting and unrelated-casting. Despite these compiler patches, Solidity developers can still make bad-casting by converting a contract type to `address` first [43].

Our tool reports bad-casting by a simple type analysis using def-use and class hierarchy information. As Table 1 shows, the tool detected 864 contracts with 1,803 bad-casting without using the `address` type. We chose 200 contracts in the order of the number of total transactions; the total number of their transactions is 1,937,026. We manually investigated them to identify exploitable contracts that have type conversions causing unexpected behaviors like calling absent functions. ***Our investigation showed that 114 out of 200 contracts are exploitable, which amounts to 57.00%. The 114 exploitable contracts consist of 33 down-casting, 73 unrelated-casting, and eight both down-casting and unrelated-casting ones. The exploitable contracts are popular, because the total number of their transactions is 730,820, and 73 contracts have more than 100 token holders.***

While investigating them, we observed two interesting points. First, among 200 live contracts, we found that only three contracts validate whether type conversion results are correct. Consider one of the example contract with validation:

```
1  contract AuctionStorage is ... {
2    bool public isAuctionStorage = true;
3    ... }
4  contract SaleClockAuction is ClockAuction {
5    function SaleClockAuction(address addr, ...) {
6     require(AuctionStorage(addr).isAuctionStorage());
7      ... } ... }
```

The `AuctionStorage` contract declares a `public` storage variable called `isAuctionStorage` on line 2. Because Solidity generates a getter function for every `public` storage variable, it generates a getter function for `isAuctionStorage` as well. The `SaleClockAuction` function on line 5 takes a parameter named `addr` of type `address`. Then, `addr` is converted to `AuctionStorage` on line 6, and the conversion result is immediately verified by calling the getter function of `isAuctionStorage`.

Second, even with the Solidity compiler patches to avoid bad-casting, developers still can perform bad-casting using the `address` type. Out of 55,046 live contracts, 21,539 contracts use type conversion; among 21,539 contracts, only 864 contracts perform type conversion without using `address`, and the remaining 20,675 contracts, which amounts to 95.99%, use casting with `address`. Our investigation implies that instead of validating type conversion results in source code, developers rely on correct inputs from external users, which is an insecure programming style.

5.5 Inheritance Order Confusion

Solidity supports multiple inheritance using the reverse of the C3 linearization order [7] to decide which function to call. Unfortunately, all the Solidity official documentation released before May 16, 2018 wrongly stated that Solidity uses the C3 linearization order. This documentation error is critical because program semantics are different depending on whether C3 linearization is used or its reverse is used. If developers rely on what the Solidity documentation stated and implement their contracts with the stated semantics, the contracts may not operate as what the developers expected. Thus, the Solidity team fixed the documentation in v0.5.0 [39, 41].

Our tool reports potentially vulnerable contracts due to inheritance order confusion when they satisfy all the following three conditions: 1) they implement multiple inheritance, 2) more than one parent contract have functions with the same signature, and 3) these parent contracts do not inherit each other. If multiple parent contracts that do not inherit each other have functions with the same signature but different semantics, then using C3 linearization and using its reverse may make different results.

As Table 1 shows, the tool detected 475 vulnerable contracts due to inheritance order confusion out of 25,417 contracts deployed earlier than May 16, 2018. We manually examined 177 contracts that have more than or equal to 10 transactions. ***Our investigation showed that only 11 out of 177 contracts may have different semantics depending on the use of C3 linearization or its reverse, which amounts to 6.21%. The total number of transactions of 11 contract is 4,336.***

During manual investigation, we observed three cases of semantics changes due to the confusion of inheritance order. The first case is *Wrong Token Minting*, where different amounts of tokens were minted because of the different semantics in the functions with the same signature in different parent contracts. We found seven contracts in this case. The second case is *Wrong Owner Checking*, where the results of owner checking were different because different objects were referenced in the functions in different parent contracts. We found three contracts in this case. Finally, the third case appeared in only one contract is *Wrong Storage Updating*. In this case, the functional logics of the functions in different parent contracts were the same, but they updated different storage variables due to variable shadowing.

Note that even though our empirical study with contracts available from Etherscan showed that only 6.21% of the contracts are vulnerable to inheritance order confusion, this vulnerability can happen in complex contracts and detecting it is quite challenging. Indeed, such bugs required extensive audits by external auditors even for one of the renowned blockchain companies [53].

5.6 Uninitialized Storage Pointers

In Solidity, if developers do not specify the `memory` keyword when they declare local variables of type `arrays` or `structs`, the local variables are by default considered as storage pointers, which are variables pointing to storage locations. Because storage is not dynamically allocated, storage pointers should be initialized before being used. If a storage pointer is uninitialized, it points to the storage slot 0 by default [45]. This semantics implies that if a developer writes some value to an uninitialized storage pointer, the value is stored in the first storage variable. Because this semantics is unintuitive, contract developers can easily make mistakes, which may lead to security issues. As a fix to this vulnerability, starting from v0.5.0, the Solidity compiler throws an error if it detects any use of uninitialized storage pointers.

Our tool reports uninitialized storage pointer uses via a simple def-use analysis. As Table 1 shows, the tool detected 89 contracts, among which 32 contracts have more than or equal to 10 transactions. We considered contracts that use uninitialized storage pointers and corrupt storage values as exploitable. ***We found that 19 out of 32 contracts were exploitable, which amounts to 59.38%. The total number of transactions of 19 contracts is 139,951, and 11 contracts have more than 100 token holders.***

We report three kinds of security issues due to this vulnerability.

Pollute winning numbers in storage (Game): Game contracts should generate winning numbers randomly, and reward users when they correctly specify the winning numbers. If users do not specify the winning numbers correctly, the contract owners get the betting fee. However, the code below shows a case where no users can win the play because of uninitialized storage pointers:

```
1  contract CryptoRoulette {
2    uint256 private secretNumber;
3    function play(uint256 number) payable public {
4      require(msg.value >= betPrice && number <= 10);
5      Game game;
6      game.player = msg.sender;
7      game.number = number;
8      if (number == secretNumber) {
9        msg.sender.transfer(this.balance); // win!
10     } ... } }
```

It generates a secret number and stores it in the `secretNumber` storage variable defined on line 2. While it seems to be a benign contract, because `game` on line 5 is an uninitialized storage pointer, two assignments of `msg.sender` and `number` on lines 6—7 store them in storage. It is a serious security issue because the value of `secretNumber` on line 2 is changed to the value of `msg.sender` since the uninitialized storage pointer `game` points to the 0th-index of storage. Therefore, to win the game, users have to input their addresses. However, because of the `require` statement on line 4, no one can win the play. The statement requires that the user input be equal or smaller than 10. Since users' unique addresses are always greater than 10, the execution terminates on line 4.

Pollute dynamically decided storage: Some contracts had uninitialized storage pointers that point to variables whose values are dynamically determined at run time by external inputs. For example, consider the following real-world contract:

```
1  contract EmojiToken is ERC721 {
2    uint256 private startingPrice = 0.001 ether;
3    uint256 private firstStepLimit =  0.063 ether;
4    address public ceoAddress;
5    function createEmojiStory(uint[] parts) public {
6      MemoryHolder storage memD;
7      for (uint i = 0; i < parts.length; i++) { ...
8        memD.bal[parts[i]] = val[parts[i]]) ... ;
9      } memD.used[parts[i]]++;
10   }}
```

The `createEmojiStory` function has an uninitialized storage pointer defined on line 6. Depending on the input value of `parts`, different storage variables can be overwritten. This is a serious security issue because the values of `parts` is controlled by external users. All the sensitive storage variables such as `startingPrice`, `firstStepLimit`, and `ceoAddress` can be overwritten.

Pollute other info in storage: We also discovered that some contracts using uninitialized storage pointers overwrite the following sensitive data:

(1) tokens: token names, total supply token numbers, and token balances of users
(2) addresses: owners of contracts and addresses of proxy contracts
(3) sales: duration of sales and refund dates
(4) prices: the maximum profit and the limit level of exchanges

While the following contract received and sent only four transactions, we report it because it leveraged uninitialized storage pointers to perform malicious behaviors:

```
1  contract ETH_ANONIM_TRANSFER {
2    uint256 feePaid;
3    uint256 creatorFee = 0.001 ether;
4    modifier secure {
5      Transfer LogUnit;
6      LogUnit.timeStamp = now;
7      LogUnit.currContractBallance = this.balance;
8      _; }
9    function MakeTransfer(address _adr, uint256 _am)
10     payable secure {
11       creator.send(creatorFee);
12       _adr.send(_am);
13   } }
```

This contract always gives financial benefits to the owner of the contract because of the uninitialized storage pointers on lines 5–7. Because the `creatorFee` is updated to `this.balance`, which is the total amount of Ether the contract reserved, the `send` function on line 12 fails since all the available Ether was sent to the owner of the contract on line 11. The failure on `send` function does not throw any exception but just returns `false` [38]. Hence, the contract owner always gains the financial profit even the `send` fails on line 12. As this example illustrates, attackers can use uninitialized storage pointers to make malicious contracts that look benign. The Solidity patch cannot prevent such malicious contracts as attackers can simply use older versions of the compiler to implement the malicious contracts.

5.7 Typo of the += Operator

Solidity used to provide the =+ operator, but it prohibits the operator from v0.5.0 because developers often make mistakes because of it. This vulnerability is the case where a developer's intention is to sum up numbers using the += operator, but accidentally used =+ as a typo. To avoid such mistakes, the Solidity compiler throws an error if it detects the =+ operator since v0.5.0.

Our tool searched for the =+ operator and found four contracts that used the =+ operator. All four contracts have more than or equal to 10 transactions, and we confirmed that they all implemented the same token named `HackerGold`:

```
1  function transferFrom(address from, address to,
2            uint256 value) ... {
3    // do the actual transfer
4    balances[from] -= value;
5    balances[to] =+ value;
6  }
```

Unlike the valid subtraction on line 4, the addition on line 5 shows the typo of +=. Such a vulnerability can be avoided if developers uses the latest compiler since it automatically prevents the vulnerability.

5.8 Use of Deprecated Functions

Using deprecated functions may lead to security issues especially when the functions were deprecated due to security problems. We inspected deprecated functions in Solidity from its release notes [13] and the official documentation [45], and found the following six deprecated functions all deprecated in v0.5.0:

- The `callcode` function was replaced with `delegatecall` because the implementation of `callcode` had bugs so that it did not preserve the correct values of `msg.sender` and `msg.value` [40, 48].

- The `throw` function is replaced with `revert` because it uses up any remaining gases, whereas `revert` returns the remaining gases to users. [40, 49]
- The `block.blockhash` function was replaced with `blockhash` because its name was misleading [36].
- The `msg.gas` function was replaced with `gasleft` because its name was misleading [37].
- The `sha3` function was replaced with `keccak256` because its name was ambiguous [35].
- The `suicide` function was replaced with `selfdestruct` to use less connotative word [11].

Among 55,046, 10,782 contracts used deprecated functions, most of which are deprecated due to their ambiguous and misleading names rather than security issues.

6 DISCUSSION

6.1 Recommendations

We give the following recommendations to Solidity developers:

- Use at least the Solidity compiler version v0.5.0 as most security patches are available from this version.
- When reusing a standard example contract, update the names of both the contract and its constructor correctly. Moreover, be careful not to make typos in constructor names. It is also risky to use `public` functions for initialization.
- Do not use the same name for storage variables in parent and child contracts. This makes the code complex to understand and possibly causes various security issues as described in this paper.
- Verify type conversion results, and do not use the `address` type for conversion because the compiler cannot check bad-casting.
- Be careful when multiple parent contracts have functions with the same signature, since different functions may be called depending on whether C3 linearization or its reverse is used.
- Do not use uninitialized storage pointers, which may corrupt storage values. While the latest Solidity compiler prevents this issue, attackers may leverage uninitialized storage pointers to implement malicious behavior.
- Use the latest Solidity compiler, which resolves several vulnerabilities such as functions without visibility, the =+ operator, and deprecated functions.

We give the following recommendations to the Solidity team:

- Release more security patches for known vulnerabilities.
- Improve the patches for constructor misuses and casting to arbitrary contracts, because they are not sufficient as we discussed.

6.2 Threats to Validity

Because we manually investigated potentially vulnerable contracts to identify developers' mistakes, the results may be subjective to human who analyzed. To reduce this threat, we analyzed contracts multiple times independently.

Since identifying exploitable contracts simply by inspecting source code is difficult, the results may be dependent on human who inspected. To lessen this threat, we additionally performed dynamic simulation using Remix [14] for complex contracts to double-check whether the contracts are exploitable. We also requested CVE IDs for exploitable contracts to receive external reviews.

7 RELATED WORK

Recent work has studied various security vulnerabilities in smart contracts. Atzei et al. [6] organized previously known vulnerabilities on Ethereum contracts. They reported vulnerabilities caused by the Solidity language features. We also discussed known vulnerabilities but we focused on those that are patched by the Solidity compiler. Delmolino et al. [10] reported common mistakes by undergraduate students from a cryptocurrency laboratory such as logic flaws in cryptography. On the other hand, we studied mistakes of real-world Solidity developers due to Solidity vulnerabilities that are patched or planned to be patched.

To detect vulnerable smart contracts, researchers studied static analysis tools. Luu et al. [27] developed a static analyzer using a symbolic executor of EVM bytecode. Kalra et al. [23] converted Solidity contract code to LLVM bitcode and detected potential vulnerabilities in contracts by using a symbolic model checker and an abstract interpreter developed for LLVM bitcode. Tsankov et al. [50] developed a domain-specific verifier, which analyzes EVM bytecode. Krupp and Rossow [25] developed an automatic exploit generation tool for smart contracts, Kolluri et al. [24] developed ETHRACER, which detects event-ordering bugs in smart contracts, and Nikolić et al. [29] introduced MAIAN, which applies inter-procedural symbolic analysis to detect vulnerable contracts. Ferreira Torres et al. [17] developed OSIRIS, which uses symbolic execution and taint analysis to detect integer bugs in smart contracts. Researchers also have formally verified smart contracts. Hirai [21] defined EVM in a language that can be compiled for multiple interactive theorem provers, and Mavridou and Laszka [28] designed and implemented a formal, finite-state machine for smart contracts. Hildenbrandt et al. [20] defined the semantics of EVM bytecode in the K framework [33], and Grishchenko et al. [18] defined a complete small-step semantics of EVM bytecode. These tools may serve as a firm ground for understanding and reasoning about the Solidity semantics. However, none of them studied most vulnerabilities discussed in this paper such as *state variable shadowing confusion*. Moreover, it is unclear whether such efforts from academia indeed improved the security of smart contracts in reality.

8 CONCLUSION

We conducted an empirical study on Solidity patches and 55,046 live contracts to understand the current security status of real-world smart contracts. Our investigation results showed that many Solidity developers are unaware of the importance of Solidity patches. We reported that 98.14% of 55,046 live contracts did not apply Solidity patches for known vulnerabilities. Furthermore, we discovered that 13,943 contracts are potentially vulnerable because Solidity patches are not adopted. Our manual investigation on potentially vulnerable contracts in source code level revealed insecure coding practices from developers and limitations of some of the Solidity patches. We found hundreds of exploitable vulnerable contracts and about one fourth of them are used by thousands of people. We hope that our empirical study results improve the security awareness of Solidity developers and help them implement secure smart contracts by preventing insecure coding practices. Finally, our work can be also helpful to security experts for security auditing of contracts and to the Solidity team for improving Solidity.

REFERENCES

[1] 2017. *Parity Wallet Hacking.* Retrieved August 23, 2019 from https://www.theregister.co.uk/2017/11/10/parity_280m_ethereum_wallet_lockdown_hack

[2] 2018. *The DAO.* Retrieved August 23, 2019 from https://en.wikipedia.org/wiki/The_DAO_(organization)

[3] 2018. *Ethereum: A Secure Decentralised Generalised Transaction Ledger.* Retrieved August 23, 2019 from https://ethereum.github.io/yellowpaper/paper.pdf

[4] 2018. *List of DApp.* Retrieved August 23, 2019 from https://dappradar.com/

[5] 2018. *A Next-Generation Smart Contract and Decentralized Application Platform.* Retrieved August 23, 2019 from https://github.com/ethereum/wiki/wiki/White-Paper

[6] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *In Proceedings of the 6th Principles of Security and Trust.* Springer, 164–186.

[7] Kim Barrett, Bob Cassels, Paul Haahr, David A Moon, Keith Playford, and P Tucker Withington. 1996. A Monotonic Superclass Linearization for Dylan. *ACM SIGPLAN Notices* 31, 10 (1996), 69–82.

[8] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanellai-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security.* ACM, 91–96.

[9] ConsenSys. 2016. *Token.sol.* Retrieved August 07, 2019 from https://github.com/ConsenSys/Token-Factory/blob/master/contracts/Token.sol

[10] 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 *In Proceedings of the 2016 Financial Cryptography and Data Security.* Springer, 79–94.

[11] Ethereum EIPs. 2015. *Renaming SUICIDE Opcode.* Retrieved August 04, 2019 from https://github.com/ethereum/EIPs/blob/master/EIPS/eip-6.md

[12] Ethereum EIPs. 2019. *EIP-1470: Smart Contract Weakness Classification (SWC).* Retrieved April 24, 2019 from https://github.com/ethereum/EIPs/issues/1469

[13] Ethereum. 2019. *Release Notes of Solidity Compiler.* Retrieved April 24, 2019 from https://github.com/ethereum/solidity/releases

[14] Ethereum. 2019. *Remix.* Retrieved July 29, 2019 from https://github.com/ethereum/remix

[15] Ethereum. 2019. *TokenERC20 Example.* Retrieved August 16, 2019 from https://github.com/ethereum/ethereum-org/blob/master/solidity/token-erc20.sol

[16] Etherscan. 2019. *Etherscan.* Retrieved July 3, 2019 from https://insights.stackoverflow.com/survey/2019

[17] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. OSIRIS: Hunting for Integer Bugs in Ethereum Smart Contracts. In *In Proceedings of 34th Annual Computer Security Applications Conference (ACSAC'18), San Juan, Puerto Rico, USA, December 3-7, 2018.*

[18] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *In Proceedings of the 7th Principles of Security and Trust.* Springer, 243–269.

[19] NCC Group. 2019. *The Decentralized Application Security Project.* Retrieved April 24, 2019 from https://dasp.co

[20] 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.

[21] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *In Proceedings of the 2017 Financial Cryptography and Data Security.* Springer International Publishing, 520–535.

[22] Yajin Zhou Xuxian Jiang and Zhou Xuxian. 2013. Detecting Passive Content Leaks and Pollution in Android Applications. In *In Proceedings of the 20th Network and Distributed System Security Symposium (NDSS).*

[23] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In Proceedings of the 25th Network and Distributed System Security Symposium (NDSS).

[24] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the Laws of Order in Smart Contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM, 363–373.

[25] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18). Baltimore, MD: USENIX Association.*

[26] V. I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (Feb. 1966), 707.

[27] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 254–269.

[28] Anastasia Mavridou and Aron Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *In Proceedings of the 22nd Financial Cryptography and Data Security.* Springer International Publishing.

[29] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the Greedy, Prodigal, and Suicidal Contracts at Scale. In *In Proceedings of the 34th Annual Computer Security Applications Conference.* ACM, 653–663.

[30] OpenZeppelin. 2019. *OpenZeppelin.* Retrieved August 16, 2019 from https://openzeppelin.com/

[31] OpenZeppelin. 2019. *SimpleToken Example.* Retrieved August 16, 2019 from https://github.com/OpenZeppelin/openzeppelin-contracts/blob/eda63c75c45e96e10d3fc188f717c8e7be64d420/contracts/examples/SimpleToken.sol

[32] OWASP. 2019. *Testing for Race Conditions.* Retrieved August 07, 2019 from https://www.owasp.org/index.php/Testing_for_Race_Conditions_(OWASP-AT-010)

[33] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[34] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *In Proceedings of the 21th Financial Cryptography and Data Security.* Springer, 478–493.

[35] Solidity. 2016. *Alias A New Name to the SHA3 Opcode.* Retrieved August 04, 2019 from https://github.com/ethereum/solidity/issues/363

[36] Solidity. 2017. *Move blockhash from block.blockhash to Global Level.* Retrieved August 04, 2019 from https://github.com/ethereum/solidity/issues/2970

[37] Solidity. 2017. *Move msg.gas to Global Level As gasleft().* Retrieved August 04, 2019 from https://github.com/ethereum/solidity/issues/2971

[38] Solidity. 2018. *Low Level Operations : send Function.* Retrieved August 07, 2019 from https://solidity.readthedocs.io/en/v0.5.7/control-structures.html?highlight=send#error-handling-assert-require-revert-and-exceptions

[39] Solidity. 2018. *Solidity Multiple Inheritance Resolution.* Retrieved August 03, 2019 from https://github.com/ethereum/solidity/issues/3856

[40] Solidity. 2018. *Solidity Official Documentation : Break Changes.* Retrieved August 22, 2019 from https://solidity.readthedocs.io/en/v0.5.7/050-breaking-changes.html#

[41] Solidity. 2018. *Solidity Official Documentation : C3 Linearization.* Retrieved August 17, 2019 from https://solidity.readthedocs.io/en/v0.5.0/contracts.html?highlight=c3%20linearization#inheritance

[42] Solidity. 2018. *Type Conversion.* Retrieved August 07, 2019 from https://solidity.readthedocs.io/en/v0.5.7/types.html?highlight=conversion#conversions-between-elementary-types

[43] Solidity. 2018. *Type Conversion Work Around.* Retrieved August 07, 2019 from https://solidity.readthedocs.io/en/v0.5.7/050-breaking-changes.html

[44] Solidity. 2019. *Ethereum Smart Contract Best Practices.* Retrieved April 24, 2019 from https://consensys.github.io/smart-contract-best-practices/known_attacks/

[45] Solidity. 2019. *Official Solidity Documentation.* Retrieved April 24, 2019 from https://solidity.readthedocs.io/en/v0.5.7/

[46] Solidity. 2019. *Shadowing of Inherited State Variables Should Be An Error.* Retrieved July 07, 2019 from https://github.com/ethereum/solidity/issues/2563

[47] Solidity. 2019. *Smart Contract Weakness Classification and Test Cases.* Retrieved April 24, 2019 from https://smartcontractsecurity.github.io/SWC-registry/

[48] Ethereum StackExchange. 2016. *Difference Between CALL, CALLCODE and DELEGATECALL.* Retrieved August 22, 2019 from https://ethereum.stackexchange.com/questions/3667/difference-between-call-callcode-and-delegatecall

[49] Ethereum StackExchange. 2017. *Why Do 'throw' and 'revert()' Create Different Bytecodes?* Retrieved August 22, 2019 from https://ethereum.stackexchange.com/questions/20978/why-do-throw-and-revert-create-different-bytecodes

[50] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *In Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security.* ACM, 67–82.

[51] Robert A Wagner and Michael J Fischer. 1974. The String-to-String Correction Problem. *Journal of the ACM (JACM)* 21, 1 (1974), 168–173.

[52] Zeppelin-Solidity. 2017. *Ownable.sol.* Retrieved August 07, 2019 from https://github.com/aragon/zeppelin-solidity/blob/master/contracts/ownership/Ownable.sol

[53] Qi Zhou. 2018. QuarkChain: Issues Due to the Inheritance Order. Personal communication.