RESEARCH-ARTICLE

# Empirical Study of Move Smart Contract Security: Introducing MoveScan for Enhanced Analysis

**SHUWEI SONG**, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

**JIACHI CHEN**, Sun Yat-Sen University, Guangzhou, Guangdong, China

**TING CHEN**, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

**XIAPU LUO**, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

**TENG LI**, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

**WENWU YANG**, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

**View all**



**Published:** 11 September 2024

**Citation in BibTeX format**

ISSTA '24: 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis
*September 16 - 20, 2024*
*Vienna, Austria*

**Conference Sponsors:**
SIGSOFT

# Empirical Study of Move Smart Contract Security: Introducing MoveScan for Enhanced Analysis

Shuwei Song
University of Electronic Science and Technology of China, China

Jiachi Chen*
Sun Yat-sen University, China

Ting Chen*
University of Electronic Science and Technology of China, China

Xiapu Luo*
The Hong Kong Polytechnic University, China

Teng Li
University of Electronic Science and Technology of China, China

Wenwu Yang
University of Electronic Science and Technology of China, China

Leqing Wang
University of Electronic Science and Technology of China, China

Weijie Zhang
Jiangsu University of Science and Technology, China

Feng Luo
The Hong Kong Polytechnic University, China

Zheyuan He
University of Electronic Science and Technology of China, China

Yi Lu
BitsLab, Singapore

Pan Li
MoveBit, China

## Abstract

Move, a programming language for smart contracts, stands out for its focus on security. However, the practical security efficacy of Move contracts remains an open question. This work conducts the *first comprehensive* empirical study on the security of Move contracts. Our initial step involves collaborating with a security company to manually audit 652 contracts from 92 Move projects. This process reveals *eight* types of defects, with half previously unreported. These defects present potential security risks, cause functional flaws, mislead users, or waste computational resources. To further evaluate the prevalence of these defects in real-world Move contracts, we present `MoveScan`, an automated analysis framework that translates bytecode into an intermediate representation (IR), extracts essential meta-information, and detects all eight defect types. By leveraging `MoveScan`, we uncover 97,028 defects across all 37,302 deployed contracts in the Aptos and Sui blockchains, indicating a high prevalence of defects. Experimental results demonstrate that the precision of `MoveScan` reaches 98.85%, with an average project analysis time of merely 5.45 milliseconds. This surpasses previous state-of-the-art tools `MoveLint`, which exhibits an accuracy of 87.50% with an average project analysis time of 71.72 milliseconds, and `Move Prover`, which has a recall rate of 6.02% and requires manual intervention. Our research also yields new observations and insights that aid in developing more secure Move contracts.

*Corresponding authors.


*ISSTA '24, September 16–20, 2024, Vienna, Austria*

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680391

## CCS Concepts

• **Software and its engineering** → **Software defect analysis**; **Domain specific languages**.

## Keywords

Smart contract, Move language, Defect, Program analysis

**ACM Reference Format:**
Shuwei Song, Jiachi Chen, Ting Chen, Xiapu Luo, Teng Li, Wenwu Yang, Leqing Wang, Weijie Zhang, Feng Luo, Zheyuan He, Yi Lu, and Pan Li. 2024. Empirical Study of Move Smart Contract Security: Introducing MoveScan for Enhanced Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680391

## 1 Introduction

Recently, Move has gained significant attention for its focus on *security-first* objective [40]. It introduces several innovations aimed at bolstering asset security, e.g., a novel data type specifically designed for this purpose, strict access controls enforced by Move modules[1], and native security component, namely `Move Prover` [73] (see §2 for details). At the time of writing, Move has been adopted by multiple blockchain platforms, such as Starcoin [58], Aptos [18], and Sui [61], with a combined market capitalization surpassing US$ 7.37 billion (as of April 6, 2024) [12].

However, the practical security efficacy of Move modules in real-world applications remains uncertain. Despite numerous empirical studies and surveys on Solidity smart contracts [64–68], to the best of our knowledge, there appears to be few studies specifically focusing on Move modules. Furthermore, although several methodologies [5, 31, 49, 73] have been proposed for detecting defects in Move modules or conducting their formal verification, there exists a significant research gap in examining the prevalence of these

[1]In this paper, the term Move modules is equivalent to smart contracts in other blockchains, and the two terms are used interchangeably.

defects in a substantial number of real-world Move modules and the identification of other potential defects.

To fill the gap, this paper presents the first empirical study on the security of Move modules. Specifically, we aim to answer the following four research questions (RQs).

***RQ1: What kinds of defects are present in Move modules?***

To uncover the real security issues, we first collect source code from as many real-world Move projects as possible. Our collection encompasses 92 projects across six key domains, i.e., DeFi, Token, Bridge, Library, Infrastructure, and Others. Then, we conduct a manual audit of 652 modules from these projects in collaboration with Movebit [41], a team specializing in blockchain security, to identify potential defects. Our audit identifies 283 modules with defects and 1,064 defects across eight distinct types. These defects pose potential security risks, cause functional flaws, mislead users, or lead to gas waste. Notably, four defect types (i.e., *unchecked return*, *infinite loop*, *unnecessary bool judgment*, and *unused constant*) represent new findings that have not been previously reported in existing studies on Move. Insights from RQ1 can inform developers and practitioners about Move's security landscape, facilitating the avoidance of defects in application development.

***RQ2: How effective is the formal verification tool in identifying defects in Move modules?***

The objective of RQ2 is to investigate the effectiveness of `Move Prover`, a formal verification tool officially launched by Move [73], in identifying the defined defects. To this end, we apply it to analyze all 283 contracts with defects. The evaluation highlights several limitations of `Move Prover` (detailed in §5.1): (1) Only 6.02% of the defects are identified due to the `Move Prover`'s insufficient functionality. (2) Auditors must draft formal specifications in a new language [73], which introduces a learning curve to be overcome. As a result, `Move Prover` has a relatively low level of automation, and developers seldom utilize it. (3) It only accepts the project source code as input and cannot analyze individual modules.

***RQ3: How effective is the existing automated detection tool in identifying defects in Move modules?***

RQ3 aims to determine the effectiveness of `MoveLint` [5], the only automated vulnerability detection tool for Move, in identifying defects. To answer RQ3, we apply `MoveLint` to the 652 modules collected in RQ1. The experimental results show that `MoveLint` successfully detects only 4.61% (49 out of 1,064) of the defects due to the inherent limitations of the compiler it depends on (detailed in §5.2). Furthermore, the tool's practicality for large-scale use is also a concern, as it, similar to `Move Prover`, necessitates access to the source code, while most modules are not open source.

***RQ4: What is the prevalence of the defects in real-world deployed Move modules?***

In RQ1, we discovered that defects do exist, which motivates us to further investigate RQ4. However, the findings from RQ2 and RQ3 underscore the limitations of existing tools in terms of accuracy, efficiency, the needs of source code, and the dependency on extensive manual efforts. These limitations make them unavailable for large-scale analysis. Consequently, RQ4 underscores the need for a bytecode analysis tool, especially given that Move modules are deployed and executed as bytecode on blockchains.

To this end, we present `MoveScan`, an automated analysis framework designed specifically for Move bytecode. Since it analyzes bytecode, `MoveScan` can be applied to deployed and undeployed modules. `MoveScan` first transforms all stack operations in the bytecode into operations on local variables, thus generating so-called *stackless bytecode* [73]. This transformation enables `MoveScan` to operate independently from the detectors. Subsequently, it extracts the necessary metadata from the stackless bytecode, i.e., control flow, function call, data dependency, and variable range information, which are then used by detectors developed upon the `MoveScan` framework. For defects that span multiple modules, `MoveScan` constructs a cross-module function call graph. This facilitates a comprehensive cross-module analysis, enabling `MoveScan` to detect defects that are challenging to identify when modules are analyzed individually.

`MoveScan` distinguishes itself from existing tools in four aspects. *Firstly*, `MoveScan` analyzes bytecode, eliminating the need for source code, which broadens its applicability. *Secondly*, `MoveScan` extracts and encapsulates metadata from the tested module, offering interfaces that streamline the creation of defect detectors for users. This design allows supporting new defect types with a few lines of code (96 on average). *Thirdly*, `MoveScan` can automatically identify the eight defect types found during audits. *Fourthly*, extensive experimental evaluations reveal that `MoveScan`'s precision (98.85%) and efficiency (averaging 0.77 milliseconds per module) significantly surpass those of existing tools. Utilizing `MoveScan`, we uncover 97,028 defects across all 37,302 modules deployed on Aptos and Sui, the two predominant blockchains utilizing Move. This finding underscores the extensive prevalence of defects in practical applications.

In summary, the main contributions of this paper are as follows:

- We conduct a comprehensive audit of 652 modules across 92 real-world Move projects, leading to the identification of eight distinct defects, including four previously unreported.
- We introduce an automated analysis framework, `MoveScan`, tailored for Move bytecode. Based on `MoveScan`, we develop eight detectors for the eight types of identified defects.
- The experimental results show that `MoveScan` performs better than existing tools in accuracy and efficiency. We apply `MoveScan` to analyze 37,302 real-world modules, and the detectors on `MoveScan` reveal 97,028 defects. We published datasets and `MoveScan` at https://anonymous.4open.science/r/support_material-D28C/.

## 2 Background

### 2.1 Move Modules

In Move, the term "*module*" **is equivalent to** the well-known concept of a "*smart contract*" [7]. The source code of modules is typically written in high-level programming languages. Post-development, a compiler translates the source code into bytecode, which is then stored and executed on the blockchain. Specifically, Move modules, developed in the Move language, are compiled into Move bytecode for execution in the Move Virtual Machine (Move VM) [7]. Blockchain users can invoke these deployed modules through transactions that specify the module's execution parameters. The transaction initiator incurs a cost, known as Gas [7], for the resources utilized during the execution process.

The following simple example illustrates the relationship between Move source code, bytecode, and opcode. Suppose there is a source code statement "let *num3* : u8 = *num1* / *num2*;" which computes the quotient of parameters *num1* and *num2*. After compiling,

the source code is converted into bytecode "0x0b000b011a0102", which is then interpreted and executed by the Move VM. The first two bytes, 0x0b and 0x00, are interpreted as opcode MOVELOC and number 0, respectively, and then executed, resulting in the first parameter, *num1*, being placed onto the stack. Similarly, the second parameter, *num2*, is placed onto the stack after executing bytecode 0x0b01. Subsequently, the Move VM executes bytecode 0x1a0102, representing the opcode DIV, POP, and RET. DIV takes two elements from the stack as operands and computes their quotient, which is then placed back into the stack. POP and RET are used to clear the stack and terminate execution.

## 2.2 Enhanced Security of Move

The improvements that Move brings to smart contract security are multifaceted: (1) It introduces a novel *resource* type to effectively manage digital assets. In Move, assets are designated as unique *resource* types, utilizing the type system to ensure that assets are neither duplicatable nor discardable without explicit intent. On the contrary, Solidity's integer-based asset representation is prone to reliability issues, such as token theft due to overflow [10]. (2) Move presents the *capability-based model* design pattern [62], which enables precise control over resource and operation access authority. (3) Move integrates a formal verification tool, `Move Prover` [73].

The `Move Prover` validates Move modules by checking whether they are against formal specifications [73]. It translates the bytecode and specifications into logical constraints. Then, it uses an SMT solver to check these constraints for violations [73]. This process helps to ensure that contracts behave as intended. (More in-depth Move's security enhancements can be found at [7].)

# 3 Methodology

This section outlines the data collection process and then explains how the collected data are utilized to investigate each RQ.

## 3.1 Data Collection

Fig. 1 illustrates our data collection process. Our data collection strategy includes collecting both the source code and bytecode of numerous Move modules to address the RQs.

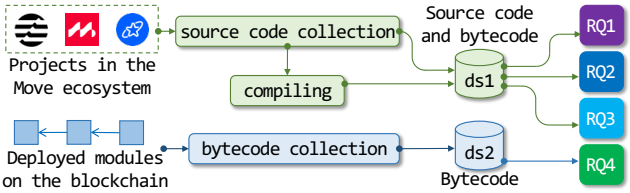


**Figure 1: Data collection process.**

*3.1.1 Dataset 1 (ds1).* To answer RQ1, RQ2, and RQ3, we create a dataset (*ds1*) of real Move projects that meet two criteria. First, it should collect source code since RQ1 requires source code for manual auditing; RQ2 and RQ3 need source code as the input of the formal verification and static detection tool. Second, the dataset should cover a wide range of *real* Move projects, as a study indicates that numerous toy contracts [34] may not accurately represent the issues found in real Move modules.

The *ds1* finally collects source code from 92 Move projects. In particular, a project typically consists of multiple modules; these projects comprise a total of 652 modules. These 92 active projects are selected from three primary sources: Aptos [18] and Sui [61], the two most popular blockchains that employ Move, and Starcoin [58], the first blockchain to employ Move. *ds1* covers six types of the most popular applications, including 41 DeFis [16], 22 tokens [4], 18 bridges [19], three libraries [39], three infrastructure [32], and five others [2]. The source code of these projects is collected from the documentation provided by Aptos foundation [17], Sui foundation [33], and GitHub repositories of Starcoin [6, 56, 59, 60].

Additionally, we utilize *ds1* as the benchmark to compare `MoveScan` with previous tools (i.e., `Move Prover` and `MoveLint`). For this purpose, we compile the source code in *ds1* into bytecode, which is necessary as `MoveScan` requires bytecode as input.

*3.1.2 Dataset 2 (ds2).* To answer RQ4, we create a second dataset (*ds2*) to store large-scale deployed modules from the blockchain. Note that modules are deployed and stored on blockchains as bytecode, meaning only the bytecode can be captured. Due to Starcoin's diminished activity, evidenced by its 24-hour transaction volume of only 716 [57]—merely 0.08% and 0.01% of Aptos's [3] and Sui's volumes [55], respectively—it does not adequately reflect the true prevalence of defects. Consequently, our bytecode collection focused on Aptos and Sui. Specifically, modules deployed on Aptos are obtained from Chainbase [29], a platform for indexing large-scale on-chain data, through the query statement "*select address, name, transaction_block_height, bytecode from aptos.move_modules*". To collect modules deployed on Sui, we begin by calling the web API provided by Suiscan [55], the block browser for Sui, to retrieve the addresses of all deployed projects. Subsequently, for each address, we use Suiscan to obtain the bytecode of all modules in that project. Through the steps outlined above, *ds2* collects 37,302 bytecode instances spanning from October 14, 2022, to January 30, 2024, with 15,479 sourced from Aptos and 21,823 from Sui.

## 3.2 Investigation Procedures for RQs

*3.2.1 For RQ1.* In collaboration with Movebit [41], a professional blockchain security company specializing in the Move ecosystem, we conduct a manual audit of the source code to identify and classify defects in Move modules. Six people contribute to the task; three of them are the authors of this paper with 1-3 years of experience in smart contract development, and the others are company employees with two years of audit experience. They analyze the 652 modules in *ds1* independently to mark the defects and related locations within the modules. In cases of inconsistent audit findings, two experts from the company with ten years of experience in cybersecurity further review the modules for detailed analysis. After this, the six participants utilize the card sorting method [54] to categorize defects. Initially, they randomly select 40% of the modules with defects and group similar defects into categories. If it is necessary to create a new categorization, then they start over by reassigning categories for all defects. They then independently categorize the remaining 60% of defects. Any disagreements are resolved through discussion between the two experts. The resulting output provides the number and category of defects for the Move project in *ds1*.

*3.2.2 For RQ2.* We evaluate whether `Move Prover` could detect defects in *ds1*. Specifically, we first try to write formal specifications for modules that contain defects. We omit modules with no defects since writing formal specifications for every module in *ds1* is

labor-intensive and time-consuming. Descriptions on writing specifications for different defects can be found in §5.1. Subsequently, we input the specification along with the project source code into `Move Prover` and detect defects by using `Move Prover` to determine if the module code conforms to the specification. Finally, the efficacy of `Move Prover` is assessed by quantifying the number of defects it successfully identifies versus those it fails to detect.

*3.2.3 For RQ3.* We use `MoveLint` to analyze the source code in *ds1*. First, we feed all 92 items into `MoveLint`, and `MoveLint` outputs details on detected defects, including their types and locations. Then, we assess the false negative (FN), false positive (FP), and precision by benchmarking against the audit results. Finally, we record the time spent by `MoveLint` analyzing each project to measure its efficiency.

*3.2.4 For RQ4.* We identify and quantify the defects present in the modules in *ds2* to answer RQ4. Given the limitations of existing tools, which rely on source code, require manual intervention, and exhibit low efficiency (as detailed in §5), their applicability in analyzing large-scale modules is limited. Hence, we propose an automated bytecode analysis framework called `MoveScan`. Building upon `MoveScan`, we create detectors for eight types of defects. The design of `MoveScan` and its detectors will be introduced in §6. Next, we utilize `MoveScan` and these detectors to analyze the bytecode in *ds1*, aiming to evaluate `MoveScan`'s effectiveness and compare it with previous tools. Subsequently, we apply `MoveScan` to analyze the bytecode of 37,302 modules in *ds2*. This analysis aims to uncover the distribution of defects in modules deployed on the blockchain and to assess `MoveScan`'s efficiency on this large-scale dataset.

## 4 RQ1: Defects in Move

By manually auditing the source code in *ds1*, we uncover 1,064 defects that can lead to security problems, functional flaws, user confusion, or increased gas consumption. The identified defects are categorized using the card sorting method (detailed in §3.2.1) with a kappa score of 0.79 [11], which shows substantial agreement of the six participants. As shown in Table 1, these defects are classified into eight distinct types. To the best of our knowledge, four of them (i.e., unchecked return, infinite loop, unnecessary bool judgment, and unused constant) have never been reported in previous work against Move [5, 31, 73]. This section introduces these defects through real-world examples that have been simplified for ease of understanding.

### 4.1 Unchecked Return (U. Return)

**Definition.** *U. Return* refers to a situation where a caller function invokes a callee function but does not receive and check the return value or only a portion of it. This defect can prevent the caller of a function from detecting unexpected states and conditions. For instance, if a function utilizes the return value to indicate success or an exception, the caller cannot handle the exception without checking the return value. Although *U. Return* has been found in C and Java, and assigned the ID CWE-252 [38], to the best of our knowledge, this issue is first discovered in Move by us.

**Example.** Listing 1 shows an example involving the *U. Return*. Line 5 defines a function *index_of()*, which receives a vector *v* and an element *e*, and returns two values. It determines the first occurrence of *e* within *v* and returns (true, index of *e*), or (false, 0) if *e* is not found. Line 3 invokes *index_of()* but neglects the first return value, resulting in possible confusion between two distinct cases. Specifically, when *_index* equals 0, it is impossible to differentiate whether *e* is absent from *v* or if *e* is the first element in *v*.

```
1 module MoveScan::unchecked_return  {
2   use 0x1::vector;
3   let (_, _index) = vector::index_of(v, e);}
4 module 0x1::vector  {
5   public fun index_of<T>(v:&vector<T>,e:&T) : (bool,u64) {...}}
6   // Returns (true,index) if the element exists in the vector;
        otherwise, returns (false,0).
```

**Listing 1: A real-world example of the U. Return.**

We advise developers to verify that the return value of each function call is properly received to fix this defect.

### 4.2 Infinite Loop (Inf. Loop)

**Definition.** *Inf. Loop* is a type of loop structure that never reaches its termination condition, resulting in repeated execution. *Inf. Loop* is not truly infinite because the blockchain employs the gas mechanism, making the *Inf. Loop* stops when the gas is depleted [7]. However, there are still three hazards to consider. Firstly, the code located after the *Inf. Loop* cannot be executed, meaning the module cannot correctly implement the expected business logic. Secondly, the performance of the blockchain is weakened as more meaningless opcodes are executed. Finally, the caller pays the gas fee but only receives an out-of-gas exception [23].

**Example.** Listing 2 shows an example involving the *Inf. Loop*. Line 2 defines a function *new()* that takes a string of bytes, duplicates it into a vector with a length of *ADDR_LENGTH*, and then returns the vector. However, the loop condition is never broken due to the programmer's neglect to update the variable that indicates the number of loops (i.e., *i* in line 4) after each iteration. Therefore, the cycle will continue until the gas is exhausted.

```
1 use 0x1::vector;
2 public fun new(bytes: vector<u8>) : vector<u8> {
3   let new_bytes = vector::empty<u8>();
4   while (i < ADDR_LENGTH) {
5     vector::push_back(&mut new_bytes, *vector::borrow(&bytes, i
          ));};
6   spec {assert false;};
7   return new_bytes}
```

**Listing 2: A real-world example of the Inf. Loop.**

We recommend ensuring variables controlling loop continuation or termination are correctly updated during the loop to fix *Inf. Loop*.

### 4.3 Unnecessary Bool Judgment (Unn. Bool)

**Definition.** *Unn. Bool* is a type of redundant statement. It is caused by evaluating boolean variables with a constant in conditional expressions and can be optimized by using the boolean variable as the conditional expression directly. This defect leads to gas waste.

**Example.** Listing 3 shows an example involving the *Unn. Bool*. Line 4 defines *is_admin()*, which verifies if *addr* is the same as the address *admin* and returns the comparison outcome as a boolean type. In line 2, the function *set()* invokes *is_admin()* and checks if the return value is *true*, which is redundant since *is_admin()* can be used directly as the conditional expression. As a consequence, the superfluous code causes avoidable gas expenses.

```
1 public fun set(addr: address) {
2   assert!(is_admin(addr) == true, ENOT_ADMIN);
3   ...}
4 public fun is_admin(addr: address) : bool {
5   return addr == @admin}}
```

**Listing 3: A real-world example of the Unn. Bool.**

**Table 1: True positives detected by different methods.**

| Method | # *U. Return* | # *Inf. Loop* | # *Unn. Bool* | # *Un. Const.* | # *Un. PF* | # *Unn. TC* | # *Ovflw.* | # *Prec. Loss* |
|---|---|---|---|---|---|---|---|---|
| Audit | 406 | 2 | 28 | 439 | 52 | 62 | 60 | 15 |
| `Move Prover` | - | 2 | - | - | - | - | 47 | 15 |
| `MoveLint` | - | - | - | - | 19 | 30 | 0 | 0 |
| `MoveScan` | 406 | 2 | 28 | 404 | 52 | 62 | 60 | 15 |

We recommend that developers eliminate *Unn. Bool* to reduce gas consumption.

## 4.4 Unused Constant (Un. Const.)

**Definition.** An *Un. Const.* is a constant that is defined but not utilized. This defect results in unnecessary gas consumption to store constants and indicates a programming bug.

**Example.** Listing 4 shows an example involving the *Un. Const.* Lines 1 and 2 define two error code constants, *AddrExist* and *AddrNotExist*. They indicate the provided address is present or absent in the whitelist. Line 3 defines *add_user_to_whitelist()*. This function adds the given address to the whitelist if it has not been previously recorded; otherwise, it throws the error code *AddrExist*. In contrast, *remove_user_from_whitelist()* in line 6 is intended for removing an address from the whitelist. An error code, *AddrNotExist*, should be thrown by this function if the address isn't present in the whitelist. However, instead of throwing *AddrNotExist* on line 7, it throws *AddrExist* due to a programmer oversight. Hence, *AddrNotExist* is defined but never utilized. This indicates a logical error where the module fails to function as intended, resulting in the generation of incorrect error codes that mislead the caller.

```
1 const AddrExist : u64 = 4;
2 const AddrNotExist : u64 = 5;
3 public fun add_user_to_whitelist(...) {
4   assert!(!table::contains<address,u64>(&mut whitelist.users,
        user_addr), AddrExist);
5   // Add user_addr to whitelist.}
6 public fun remove_user_from_whitelist(...) {
7   assert!(table::contains<address,u64>(&mut whitelist.users,
        user_addr), AddrExist);
8   // Remove user_addr from whitelist.}
9 fun get_whitelist() {// Return the whitelist.}
```

**Listing 4: An example of the Un. Const. and Un. PF.**

## 4.5 Unused Private Function (Un. PF)

**Definition.** *Un. PF*s are private functions that have been defined but not used. It is a form of dead code since these private functions will never be invoked, either inside or outside the module. This defect results in gas waste and implies that the programmer has misconfigured the function's visibility, resulting in functional flaws.

**Example.** Line 9 of listing 4 shows an *Un. PF* that is supposed to be set to public for external access to the whitelist.

We recommend that developers fix *Un. Const* and *Un. PF*. If they are unnecessary, they should be removed; otherwise, they should be properly utilized to ensure the code adheres to the business logic.

## 4.6 Unnecessary Type Conversion (Unn. TC)

**Definition.** *Unn. TC* refers to type conversions that perform even when the original and target types are identical. This defect results in the needless execution of the CAST opcode and gas waste.

**Example.** Listing 5 shows an example involving the *Unn. TC*. Line 1 defines *time_delta()*, which calculates the u64 type difference between the current timestamp and a given timestamp. Line 2 calls *now_seconds()* to obtain the current timestamp, *now_timestamp*, in u64 type. Line 3 aims to convert *now_timestamp* to u64 type, rendering this a superfluous and meaningless type conversion.

```
1 public fun time_delta(timestamp : u64) : u64 {
2   let now_timestamp = Timestamp::now_seconds();
3   return (now_timestamp as u64) - timestamp;}
```

**Listing 5: A real-world example of the Unn. TC.**

We recommend that developers eliminate *Unn. TC* to reduce gas consumption.

**Insight:** While the Solidity compiler effectively optimizes certain defects, such as *Un. Const.*, *Un. PF*, and *Unn. TC*, our comprehensive analysis reveals that the Move compiler fails to address all the defects we identified. These defects highlight the Move compiler's restricted degree of optimization and provide directions for its future improvement. Consequently, defect detection tools for Solidity can disregard defects already resolved by compilers, whereas tools for Move should account for them.

## 4.7 Overflow (Ovflw.)

**Definition.** *Ovflw.* refers to an error that occurs when a value (usually a number) exceeds the maximum value that can be represented within a given data type. Move VM has already considered *Ovflw.* caused by basic arithmetic operations: addition, subtraction, and multiplication. It checks whether the result exceeds the limit when executing these operations, and if so, it aborts the program [7]. However, we find that it ignores the SHL opcode (i.e., the left shift operation). The Move VM does not abort this type of *Ovflw.*, making it more insidious for programmers and capable of causing significant logic errors. Therefore, this paper specializes in the *Ovflw.* caused by SHL.

```
1 const INVALID_STATE : u8 = 1;
2 fun make(category: u8, reason: u64): u64 {
3   spec {assert (reason << 8) >= reason;};
4   (category as u64) + (reason << 8) }
5 public fun invalid_state(reason: u64): u64 {
6   make(INVALID_STATE, reason) }
```

**Listing 6: A real-world example of the Ovflw.**

**Example.** Listing 6 shows an example involving the *Ovflw.* Line 2 defines *make()*, which generates an error code based on two parameters, *category* and *reason*. Line 5 defines *invalid_state()*, which invokes *make()* to generate an error code specific to the category *INVALID_STATE*. The left-shift operation on variable *reason* in line 4 could cause an *Ovflw.* More specifically, *reason* is a 64-bit integer, and if its value falls within the range of $2^{56}$ and $2^{64} - 1$, performing the left-shift operation results in an overflow value between 0 and $2^8 - 1$. As a result, an incorrect code is generated, and different *reason* could produce an identical error code, causing ambiguity.

We recommend creating a library similar to Solidity's SafeMath to prevent *Ovflw.*

**Insight:** The current *Ovflw.* protection of Move VM is flawed and could be improved by adding checks on the result of SHL opcode.

### 4.8 Precision Loss (Prec. Loss)

**Definition.** The Move language does not support float types; thus, division and square root operations could result in rounded-down values. Therefore, performing the operations in the order of multiplication followed by division[2] consistently achieves precision better than or equal to the sequence of division followed by multiplication. We define the latter situation as *Prec. Loss*. This defect can result in inconspicuous data corruption and financial loss.

**Example.** Listing 7 shows an example involving the *Prec. Loss*. Line 1 defines *claim_rewards()*, which computes a user's reward based on its investment share. However, there is a *Prec. Loss* when calculating the user's share (line 2), resulting in the user not receiving any reward. In particular, the user's share is usually much smaller than the total amount in the pool (i.e., *deposit* is less than *pool.amount*), so the *Prec. Loss* causes *amount_claim* to be set to 0, with the result that the user receives no reward. To mitigate this, line 2 should allow multiplication to be executed before division.

```
1 public fun claim_rewards(deposit: u64) {
2   amount_claim = (deposit / pool.amount) * pool.share;
3   spec {
4     assert amount_claim==pool.share * deposit / pool.amount;};
5   ...}
```

**Listing 7: A real-world example of the Prec. Loss.**

We recommend that developers reorder calculations to perform multiplication before division to fix *Prec. Loss*.

**Answer to RQ1:** We identify eight distinct defect types within Move modules, each potentially causing security problems, functional flaws, user confusion, or increased gas consumption.

## 5 Effectiveness of Existing Tools

This section examines the effectiveness of `Move Prover` and `MoveLint` in detecting the aforementioned eight types of defects.

### 5.1 RQ2: Effectiveness of `Move Prover`

`Move Prover` detects defects by verifying whether the source code meets formal specifications. Therefore, it is necessary to prepare appropriate specifications in addition to the source code. Since the specification must be embedded in source code, we first check if the modules in *ds1* already contain specifications written by developers. If appropriate specifications are present, we can reuse them without having to write them from scratch. We locate specifications by searching for the keyword *spec* in source code. Unfortunately, the statistics show that only 89 modules contain specifications.

**Observation:** Move recommends developers use `Move Prover` for formal verification, but we find only 89 (13.65% for the adoption rate) modules that contained specifications in *ds1*.

Moreover, upon reading these specifications, we find that they cannot be used to detect the defined defects, as they are used for other purposes (e.g., verifying the number of tokens sent matches those received). Therefore, we must manually write specifications for each module with defects in *ds1*. `Move Prover` offers a specialized language called *Move Specification Language (MSL)* for writing specifications [20], and we finally successfully write specifications for the following three defects.

**Inf. Loop.** After each loop body, we write the specification statement "assert false". This statement throws an error when executed by `Move Prover`. To determine if there is an Inf. Loop, we can check if `Move Prover` throws an error; if an Inf. Loop occurs, "assert false" will not be executed, resulting in no error being generated. Listing 2 shows an example, with the specification written in line 6.

**Ovflw.** It is obvious that if no Ovflw. is generated, then the result of SHL (denoted by *res*) must not be less than the operand (denoted by *op*). With this observation, we set the formal specification to "assert *res* >= *op*". If `Move Prover` throws an error, it indicates that an Ovflw. has been detected. Listing 6 shows an example, with the specification written in line 3.

**Prec. Loss.** We insert specifications after each statement that involves both multiplication and division (or square root operation) to check if changing the execution order alters the result. If it does, we detect a Prec. Loss. Listing 7 provides an example, with lines 3 and 4 containing the specifications we insert.

`Move Prover` is incapable of detecting the remaining five defects by writing proper MSL specifications, as `Move Prover` works by monitoring state changes of variables, including changes to the values of base types and changes to the storage locations of resource types. However, these five types of defects do not involve state changes of variables. For instance, Unn. TC only involves the variable's type, not its value, and therefore cannot be identified.

**Insight:** `Move Prover` would benefit from adding formal specifications for variable declarations, usage, and variable types. This will facilitate the detection of a wider range of defects.

After preparing specifications, we run `Move Prover` to detect defects. As shown in Table 1, `Move Prover` only reports 64 defects. By manually checking these reported results, we determine they are all true positives (TPs). The remaining defects not reported by `Move Prover` are its FNs. Even excluding the defect types it doesn't support, `Move Prover` still has 13 FNs, which are caused by a complex bug [24] (i.e., Boogie type error with bit representation). In summary, the precision, recall, and F1-score of `Move Prover` are 100%, 6.02%, and 11.35%. Excluding the five defect types it doesn't support, its recall and F1-score improve to 83.12% and 90.78%, respectively.

**Answer to RQ2:** The `Move Prover` has limitations in identifying defects due to the requirement for human intervention, the limited number of supported defect types, the absence of bytecode support, and the necessity for complete project source code.

### 5.2 RQ3: Effectiveness of `MoveLint`

By applying `MoveLint` to *ds1*, it reports 56 defects covering only three defect types. Upon manual inspection, we discover that seven are FPs caused by `MoveLint`'s oversimplified approach to identifying Prec. Loss. It considers any statement with division followed by multiplication as Prec. Loss, without taking into account divisibility. For instance, in the operation $100/10 * 9$, 100 is divisible by 10 and

[2]The *Prec. Loss* defined in this paper considers both division and square root operations.

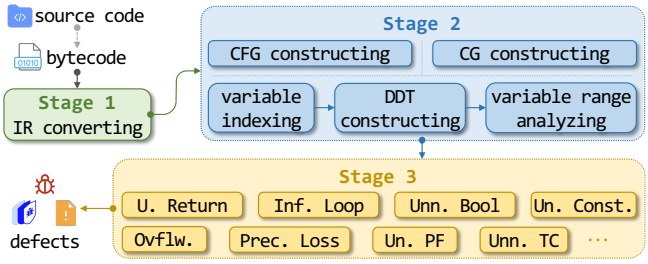


**Figure 2: Architecture of MoveScan.**

does not result in Prec. Loss. Moreover, we find 1,015 defects identified in manual audit are not detected by MoveLint. Upon scrutinizing the source code, we ascertain that they represent MoveLint's FNs.

In Table 1, "–" indicates the defect types that MoveLint does not support. Besides, we find MoveLint's notable FN rate is attributed to its limited support for only the native Move language, excluding Move variants modified by Aptos and Sui. Specifically, MoveLint utilizes Abstract Syntax Tree (AST) for defect detection [5], which relies on the native Move compiler. However, the adaptations made by Aptos and Sui (i.e., the use of inline assembly in library functions) necessitate their own new compilers and are incompatible with the native compiler. As a result, 460 modules developed in modified Move cannot be compiled by the native compiler used by MoveLint, leading to its high FN rate. Aside from the defect types and modules it does not support, we find that it has seven other cases of FNs. The reason is that it only considers constants and ignores variables when detecting Ovflw. For example, in the case shown in listing 6, *reason* is a variable, and MoveLint does not report it as an Ovflw.

**Insight:** MoveLint could be improved by adding support for analyzing individual modules. This would allow developers to test specific modules rather than entire projects.

In summary, the precision, recall, and F1-score of MoveLint are 87.50%, 4.61%, and 8.75%. If the modules it can't compile are excluded, the recall and F1-score will increase to 25% and 38.89%. MoveLint's time overhead is deemed acceptable as it analyzes projects automatically and takes an average of 71.72 milliseconds per project.

**Answer to RQ3:** MoveLint is not effective in defect identification due to its high FN rate, lack of bytecode support, and requirement for complete project source code.

# 6 RQ4: Our MoveScan Approach

## 6.1 Overview

Since none of the existing tools is competent to investigate RQ4, a new tool must be designed to meet the requirements of automation, bytecode support, more defect types support, and high precision. However, analyzing bytecode automatically and efficiently poses a challenge due to its stack-based nature, which hinders metadata extraction. Additionally, some defects involve cross-module invocations, further complicating analysis. To address these issues, MoveScan converts the bytecode into an intermediate representation (IR) in stage 1 (§6.2), which is more conducive to metadata extraction. Then, MoveScan constructs the cross-module call graph based on the IR in stage 2 (§6.3) to facilitate defect detection that necessitates examining interactions across multiple modules.

As shown in Fig. 2, MoveScan contains three stages. It takes bytecode as input and outputs the detected defects, also offering compatibility with source code by first compiling it into bytecode using a compiler. The first stage converts bytecode into IR, which facilitates the analysis by replacing stack operations with operations on local variables. The second stage takes stackless bytecode as input and collects the metadata necessary for detecting defects, including the control flow graph (CFG), call graph (CG), data dependencies tree (DDT), and variable ranges. In the final stage, eight detectors retrieve necessary metadata through the interface provided by MoveScan and subsequently identify defects. Additionally, MoveScan allows users to develop new detectors using these interfaces, enhancing its adaptability to emerging defects.

## 6.2 Stage 1: IR Converting

MoveScan converts bytecode to stackless bytecode before analysis as it is not easy to analyze bytecode directly. Fig. 3 displays the source code, bytecode (in opcode format), and stackless bytecode of function *f()*. This example highlights the advantages of using stackless bytecode for analysis over regular bytecode. The source code clearly shows that the function performs arithmetic operations on three parameters. However, if the source code is not available and only bytecode is accessible, determining this function's functionality, control flow, and data dependencies becomes challenging. For instance, to identify the dividend for the DIV (line 4 of opcode), all preceding opcodes must be executed to reconstruct the stack context at the point where DIV is executed. In other words, only by running lines 0-4 do we learn that the dividend is the product of the first two parameters. On the contrary, obtaining this information from stackless bytecode is straightforward. Specifically, line 4 of stackless bytecode indicates that the dividend is *$t5*, which relies on *$t3* and *$t4*, as shown in line 2. Note that in this example, analyzing opcode is relatively straightforward due to its brevity. However, when examining more complex contracts (e.g., *ds1* and *ds2*), the advantages of stackless bytecode become increasingly evident.

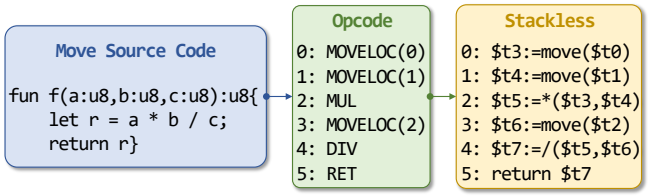


**Figure 3: Conversion of opcode to stackless bytecode.**

One method for obtaining stackless bytecode is using Move Prover, which operates in three steps: (1) It converts each module in the tested project into a *CompiledModule* data structure. This structure contains not only the bytecode but also the module ID, constant pool, and function signatures. (2) Move Prover creates a data structure called *GlobalEnv* to store the CompiledModule of all modules in the project. Move Prover uses this data structure to analyze cross-module calls. (3) Move Prover takes GlobalEnv as input and converts the module bytecode stored in it to stackless bytecode.

However, this approach has two limitations. Firstly, it does not support bytecode, since it needs to extract formal specifications from the source code in the first step. Secondly, building GlobalEnv in the second step requires information about all modules in

the project. As a result, it cannot generate stackless bytecode for individual modules, such as those deployed on the blockchain.

To facilitate the support for bytecode and individual modules, we refine `Move Prover` in two aspects. First, as our goal is not formal verification, extracting formal specifications from the source code is unnecessary. Therefore, we modify the first step of `Move Prover` to utilize the compiler's functionality in converting the target module from bytecode, rather than from source code, into a CompiledModule. Second, our practice shows that building GlobalEnv for the entire project to obtain cross-module call information is unnecessary, as the CompiledModule corresponding to each module already records the IDs of other modules that this module depends on, as well as the signatures of the external functions it calls. Accordingly, we modify the second and third steps of `Move Prover`, allowing us to extract the signature of the called function directly from CompiledModule rather than GlobalEnv, enabling the conversion of a single module bytecode into stackless bytecode.

## 6.3 Stage 2: Code Analyzing

At this stage, `MoveScan` processes stackless bytecode to extract metadata essential for further defect detection.

**CFG Constructing.** `MoveScan` constructs CFGs in two steps. First, the stackless bytecode is broken down into basic blocks (BBs), which are sequential pieces of code without any jumps or branches except at the end of the block. BBs are identified by their beginning (the first instruction of the module or the target of a jump instruction) and their end (a jump or return instruction). Next, `MoveScan` creates a node for each BB and connects them with directed edges. The edges point from the node with the jump instruction to the node targeted by the jump. These nodes and edges form a CFG that represents all possible execution paths of the module.

**CG Constructing.** The CG records function calls within a module and, if present, function calls across modules. `MoveScan` constructs CGs in four steps. First, `MoveScan` identifies all function calls (both intra-module and cross-module) by searching for *Call()* in stackless bytecode. Second, `MoveScan` distinguishes between intra-module calls and cross-module calls by parsing the third parameter of *Call()*, which contains the called module ID. A cross-module call is identified when the called module ID differs from the module under analysis, whereas an intra-module call is recognized when both IDs are the same. Third, `MoveScan` constructs CG for a module by creating nodes for each function and establishing directed edges that point from a function to the functions it calls. If the modules with functions involved in cross-module calls are also supplied to `MoveScan`, then in its fourth step, it amalgamates multiple CGs to form a comprehensive cross-module call graph. Specifically, `MoveScan` generates a CG for each module and then creates edges for cross-module calls, linking the CGs of the caller and callee.

**DDT Constructing.** `MoveScan` obtains dependencies between data in the following steps. First, `MoveScan` collects variable assignment statements from stackless bytecode and constructs an index from the variable to them. For instance, in Fig. 3, the variable *$t7* is mapped to the statement *$t7*:=/(*$t5*,*$t6*). Using this index, `MoveScan` can efficiently retrieve the composition of a given variable. Second, `MoveScan` traverses variable assignment statements, constructing

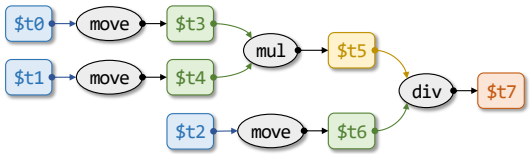


**Figure 4: An example of a data dependency tree.**

them into a data dependency tree. For instance, the stackless bytecode from Fig. 3 is transformed into the tree shown in Fig. 4. In this representation, each rectangle denotes a variable, and each ellipse signifies an operation. Directed edges from rectangle $X$ to ellipse $Y$ indicate that $X$ is an operand of $Y$, while edges from ellipse $Y$ to rectangle $Z$ denote that $Z$ is the result of $Y$.

**Variable Range Analyzing.** `MoveScan` maintains the legal range of each variable, which is the key to detecting defects such as Ovflw. The variable range is obtained in three steps. Firstly, `MoveScan` collects all variables and their types from stackless bytecode. Secondly, `MoveScan` sets the initial legal range for each variable based on its type. Lastly, `MoveScan` traverses the data dependency tree, and a variable's range is modified during its operation, including addition, subtraction, multiplication, division, modulo, bitwise operations, and type conversion. For instance, if a variable undergoes a type conversion, its range should align with that of the new type.

## 6.4 Stage 3: Defect Detecting

At this stage, our framework offers users interfaces to create detectors for identifying different defects. Each detector utilizes interfaces to retrieve essential metadata as required and then identifies defects by evaluating whether characteristics of these defects are present in the target module. Developing a detector requires an average of only 96 lines of Rust code, as `MoveScan` handles the complex task of extracting metadata, while the detector only needs to focus on the characteristics of defects. This design gives `MoveScan` good scalability. Following details eight detectors used to detect the defined eight defects, with the numbers in "< >" indicating their respective lines of code.

**U. Return** < 90 >**.** The basic idea for detecting U. Return is to identify all function calls and then check that each return value is not discarded. The detector contains the following steps. First, the detector requests the CG and stackless bytecode of the target module from `MoveScan`. It then examines each function call in the CG to determine if the called function has return values. If a return value is present, the detector checks whether a *Destroy* instruction follows the *Call()* instruction. The presence of a *Destroy* instruction indicates that the return value is discarded without being checked, signifying an U. Return. If the detector identifies that the called function has $n$ return values, it assesses whether any *Destroy* exists in the $n$ instructions following the *Call()*. The occurrence of any *Destroy* in this sequence confirms an U. Return.

**Inf. Loop** < 196 >**.** A normal loop should include exits in the form of branching statements. These statements use the value of a key variable to determine whether the loop should continue or stop. For instance, in line 4 of listing 2, the variable $i$ serves as the key variable. To detect Inf. Loop, we first identify the loop and find the key variables from the branching statements. Then, we check whether these variables are modified during the loop. If they are not modified, the loop is infinite.

This detector first requests CFG, CG, and DDT of the target module and then works through the following six steps. Initially, the detector employs fast dominance algorithms [14] to identify loops within the CFG, and each loop is represented by several basic blocks. Subsequently, the detector locates branch statements within these loops. This is done by searching for *if...else...* in stackless bytecode. The third step is determining if a branch statement serves as a loop exit. This can be identified if one of the branch's successor blocks is located outside the loop. For each loop exit, step four involves collecting the conditional expressions, that is, the expressions following *if*. Step five entails gathering key variables from the collected conditionals, which dictate the continuation or termination of the loop. It is important to also collect the variables on which these key variables depend, as these variables can indirectly control the loop's termination. The dependencies between variables can be obtained from the DDT. In the final step, the detector reviews all instructions within the loop to check if the key variables and the variables they depend on are modified. An absence of modifications across all such variables indicates an Inf. Loop.

**Unn. Bool** < 100 >**.** The basic idea is to check if there is a statement that determines whether a variable of type Bool is equal to a constant. The detector first requests the DDT and stackless bytecode of the target module from MoveScan. Then, for each *EQ* (Equal) and *NEQ* (Not Equal) instruction, the detector examines the two operands of the instruction. If one operand is a constant of Bool type and the other is a Bool variable, an Unn. Bool is identified.

**Un. Const.** < 61 >**.** The detector first requests the target module's stackless bytecode and constant pool from MoveScan. It then traverses the stackless bytecode and marks a constant if it is used. Finally, the constants that remain unmarked are labeled as unused.

**Un. PF** < 73 >**.** The detector first requests the CG of the target module from MoveScan. It then traverses all nodes (each node represents a function) in the CG, identifying those without incoming edges. Such nodes indicate that they have no callers and are unused. These functions are then evaluated for their visibility; if marked as private, they are classified as Un. PFs.

**Unn. TC** < 88 >**.** The detector first retrieves the target module's stackless bytecode and DDT from MoveScan. It then locates every type conversion instructions in the stackless bytecode (e.g., *CASTU8*, *CASTU64*, and *CASTU256*). For each instruction, the detector identifies the operand's type from the DDT and checks whether it matches the target type of the conversion. If they are identical, an Unn. TC is detected. For instance, a *CASTU8* instruction with an operand type U8 (i.e., unsigned char) would be deemed an unnecessary conversion.

**Ovflw.** < 66 >**.** Please note that this detector specifically targets Ovflw. caused by the SHL opcode (detailed in §4.7). The method for detecting Ovflw. is to check if the variable exceeds the maximum limit of its type. This detector first retrieves the target module's stackless bytecode and variable range from MoveScan. For each SHL, it then assesses whether the resulting value's range exceeds the maximum limits. To illustrate, consider an example where the SHL opcode acts on a variable *src* with a range $[m, n]$, shifting it left by one bit to produce *dst*. Under normal circumstances, *dst* should be twice the value of *src*. Therefore, MoveScan computes the range of *dst* as $[2m, 2n]$. If the maximum value that *dst*'s type can represent is smaller than $2n$, an Ovflw. risk is detected.

**Prec. Loss** < 90 >**.** Since dividing before multiplying can cause Prec. Loss, it is necessary to recognize the order in which the division and multiplication are performed to detect it. This detector first requests the target module's stackless bytecode and DDT from MoveScan. Then, for every *MUL* instruction in the stackless bytecode, it tracks the data dependencies of the two operands of *MUL*. This task requires examining the DDT to determine if any operand is the result of a division (*DIV*) or square root (*SQRT*) operation. If this is the case, it indicates a potential Prec. Loss. To prevent the occurrence of FPs, similar to MoveLint (detailed in §5.2), our detector also takes into account divisibility. It does not report a Prec. Loss when the dividend is divisible by the divisor. Completing this task is simple for us because MoveScan constructs a complete DDT that clearly shows the dividends and divisors.

## 6.5 Evaluation

*6.5.1 Precision.* To evaluate the precision of MoveScan, we input 92 projects from *ds1* into MoveScan, which analyzes a total of 652 modules within these projects and reports 1,041 detected defects. Out of the 1,064 defects discovered through the manual audit, only 35 are not detected by MoveScan. On manually inspecting the source code of modules containing these undetected defects, we determine them as FNs of MoveScan. All FNs are of type Un. Const., caused by the compiler's optimization for duplicate constants.

Listing 8 shows an example of a FN. During the investigation of RQ1, the audit reveals that the constants in lines 1 and 2 (i.e., *ERROR_A* and *ERROR_B*) are used, while those in lines 3 and 4 (i.e., *STATE_P* and *STATE_Q*) are marked as Un. Const. However, the compiler recognizes that *ERROR_A* and *STATE_P* have the same value; therefore, it only retains one when converting the source code into bytecode. The same applies to *ERROR_B* and *STATE_Q*, which are retained as one. The names of constants are not recorded in bytecode. Therefore, MoveScan only recognizes two constants, 0 and 1, of type u8. Since both 0 and 1 are used, MoveScan assumes that no Un. Const. exist. Fortunately, such Un. Const. do not pose a security risk and gas waste, so we conclude that MoveScan does not need to be improved to deal with these FNs.

```
1 const ERROR_A : u8 = 0; // used
2 const ERROR_B : u8 = 1; // used
3 const STATE_P : u8 = 0; // unused
4 const STATE_Q : u8 = 1; // unused
```

**Listing 8: An example of MoveScan's false negatives.**

```
1 fun count_leading_zeros(x: u128) {
2   if (x & 0xFFFFFFFFFFFFFFFF0000000000000000 == 0) {
3     // x's higher 64 is zero, shift the lower part over
4     x = x << 64;};}
```

**Listing 9: An example of MoveScan's false positives.**

Additionally, we examine the defects reported by MoveScan but not identified in manual auditing, and find 12 such instances. Upon meticulously inspecting the corresponding module source code, we discover that all these instances are FPs of MoveScan. These FPs are all classified as Ovflw. and an example is shown in listing 9. MoveScan reports a defect on line 4 due to the variable $x$ being shifted 64 bits to the left and then reassigned to $x$. MoveScan recognizes the value after the left shift as being out of bounds. However, since the higher 64 bits of $x$ are all zeros (as shown in line 2), there is no actual Ovflw. To address this issue, we plan to integrate symbolic

execution [37] into MoveScan to collect path constraints in future work. In summary, the precision, recall, and f1 score of MoveScan are 98.85%, 96.71%, and 97.77%.

• **Comparison with Move Prover.** The experimental results demonstrate that MoveScan surpasses Move Prover in defect detecting. The sets of defects reported by the manual audit, Move Prover, MoveLint, and MoveScan are denoted by $D_{MA}$, $D_{MVP}$, $D_{ML}$, and $D_{MS}$. As shown in Fig. 5, all defects detectable by Move Prover are also identifiable by MoveScan, whereas not all defects recognized by MoveScan are detectable by Move Prover. This is quantified as $D_{MVP} - D_{MS} = 0$ and $D_{MS} - D_{MVP} = 977$, of which only 12 are FPs of MoveScan.

• **Comparison with MoveLint.** Fig. 5 shows that $D_{ML} - D_{MS} = 7$, which are all FPs of MoveLint. On the other hand, $D_{MS} - D_{ML} = 992$, of which only 12 are FPs. This result indicates that MoveScan is more capable of detecting defects than MoveLint, as all TPs of MoveLint can also be identified by MoveScan, but not vice versa.

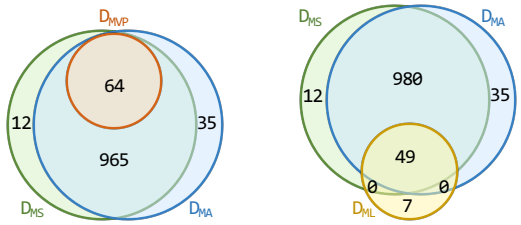


**Figure 5: Comparison of defect sets reported by MoveScan, MoveLint, and Move Prover.**

*6.5.2 Efficiency.* In *ds1*, MoveScan only requires 0.77 milliseconds to analyze one module on average, with the analysis time for 90.22% of the modules being under 16.91 milliseconds. Since MoveLint analyzes entire projects rather than individual modules, we aggregate the time taken by MoveScan to analyze each module within a project for a comparative assessment against MoveLint. The results show that MoveScan takes, on average, only 5.45 milliseconds to analyze a project in *ds1*, which is only 7.60% of the time required by MoveLint. We analyze MoveLint's source code and find that its relatively high overhead is due to calling the compiler to compile the target module before analysis. This includes steps such as optimization and IR conversion, which have a significant impact on performance.

## 7 RQ4: Prevalence of Defects

By applying MoveScan to analyze 37,302 modules in *ds2*, it identifies 97,028 defects, averaging 2.60 defects per module. The bar graph in Fig. 6 shows the frequency of each type of defect in modules deployed on the two blockchains (i.e., Aptos and Sui). The graph reveals a similar distribution of defects across both blockchains. For instance, the Un. Const. and U. Return are the most common, while the Inf. Loop occurs the least. This pattern emerges because Un. Const. and U. Return, posing no runtime exceptions, is easily overlooked, while Inf. Loop leads to significant runtime errors, making it more readily identified and rectified by programmers.

**Answer to RQ4:** Defects are commonly found in practice, with an average of 2.60 defects present in each module.

We further investigate the tendency of the number of defects over time. The data in Fig. 7 indicates a consistent increase in the number of defects. The largest rise is shown in Oct. 2022 and Apr. 2023, which correspond to the official launch of Aptos and Sui, respectively. Fig. 8 shows the number of defects in a module relative to its size. It is evident that the number of defects in a module tends to increase as the size of the module increases.

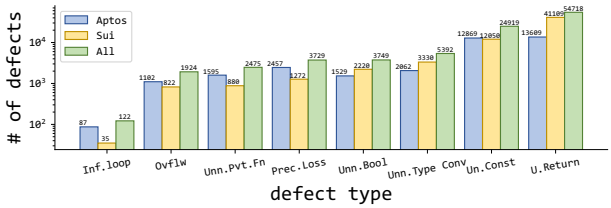


**Figure 6: Defects detected in deployed modules.**

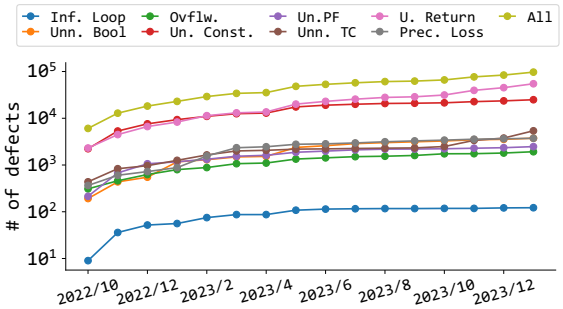


**Figure 7: Number of defects over time.**

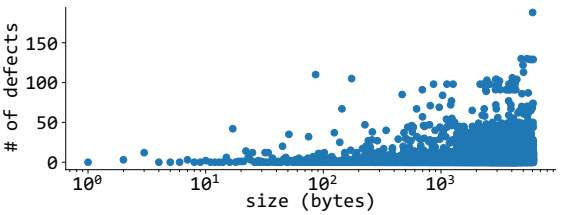


**Figure 8: Number of defects as a function of module size.**

Fig. 9 shows the CDF of the time consumed by MoveScan to analyze the modules in *ds2*, where the three lines correspond to the Aptos module, the Sui module, and their aggregation, respectively. The analysis of 90% of the modules takes no more than 3.01 milliseconds. Additionally, we observe a noteworthy phenomenon: MoveScan takes an average of 2.01 milliseconds to analyze each module in *ds2*, which is 2.61 times of the time spent analyzing modules in *ds1*. We hypothesize that this difference may be due to the larger code size of modules deployed on blockchain than those in open-source projects. To test this hypothesis, we count and compare the sizes of the modules in *ds1* and *ds2*. The statistics indicate that the module sizes in *ds2* are, on average, 1.66 times larger than those in *ds1*, confirming our conjecture.

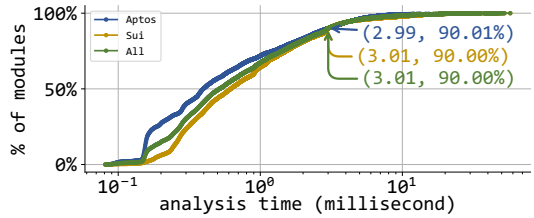


**Figure 9: CDF of analysis time per module in *ds2*.**

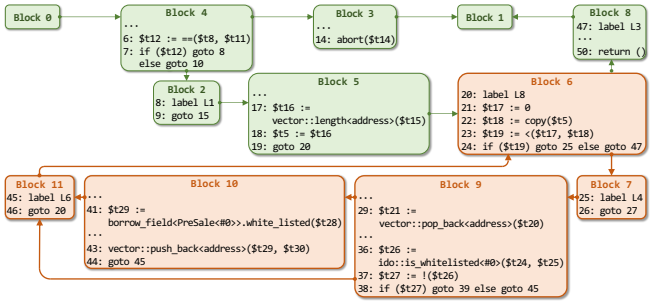


**Figure 10: The CFG of a real-world function with an Inf. Loop.**

## 8 Case Study

As shown in Fig. 10, MoveScan identifies a real-world function with an Inf. Loop and automatically generates its CFG. According to MoveScan's report, the Inf. Loop starts at block 6 of the CFG. Manual inspection confirms this report is a true positive, as blocks 6, 7, 9, 10, and 11 form an Inf. Loop. Due to the lack of a decompilation tool, we manually analyze the bytecode and infer the source code, as depicted in listing 10. The function aims to whitelist addresses from an input array (line 1). It iterates through the array (line 3), checks if each address is already whitelisted, and adds it if not (lines 5-6). However, due to developer oversight, the variable *len*, which controls the loop's continuation or termination, is not updated at the end of the loop body (line 7), resulting in an Inf. Loop. Consequently, this function cannot fulfill its intended purpose, wasting the caller's gas and, in extreme cases, reducing the blockchain's performance.

```
1 public fun add_white_list(mut vec: vector<address>) {
2     let mut len = vector::length(&vec);
3     while (len > 0) {
4         let addr = vector::pop_back(&mut vec);
5         // If addr is in the whitelist, skip it;
6         // otherwise, add it to the whitelist.
7         // The statement len = len - 1 is missed.}}
```

**Listing 10: Source code of a real Inf. Loop.**

## 9 Related Work

### 9.1 Analysis for Move

The analysis methods for Move can be categorized into three main groups, the first is formal methods [31, 43, 73]. `Move Prover`, released alongside the Move language, is a formal verification tool developed by the core Move team [33]. It verifies Move project source code against formal specifications through a multi-step process involving specification extraction, bytecode compilation, and SMT formula generation for verification by an SMT solver [73]. Enhancements to `Move Prover` include a conversational framework by Nelaturu et al. [43] for contract generation and VeriMove by Keilty et al. [31] for automatic Move contract creation based on specifications. Despite its theoretical strengths, practical limitations arise from the auditor's understanding and specification accuracy. The second method is `MoveLint`, which is currently the only static analysis tool available. It converts source code to AST and identifies vulnerabilities through predefined rules [5]. It faces challenges such as bytecode incompatibility and only native Move support (§5.2). The third method is Move Verifier, part of the Move VM, which checks bytecode before execution [7]. Its limitations include the need to modify the VM to support new vulnerabilities, and it only supports checking deployed and invoked modules. MoveScan, developed in response, offers pre-deployment module analysis and complements Move Verifier.

### 9.2 Vulnerability Detection for Solidity

**Static Analysis.** SmartGraph [74] converts Solidity code into graphics and provides a set of rules to detect reentrancy, overflow, and other vulnerabilities. EtherSolve analyzes contracts by constructing accurate CFGs [50]. The eTainter detects gas-related vulnerabilities by performing taint analysis on bytecode [21]. SmartCheck converts the source code to IR and then checks it using pattern matching [63].

**Symbolic Execution.** AChecker combines static data stream analysis and symbolic execution to detect access control vulnerabilities [22]. Liu et al. perform symbolic execution and vulnerability constraint solving on control flow [35]. Oyente examines the execution path of contracts and identifies prevalent vulnerabilities [37]. ReDetect focuses on improving the accuracy of detecting reentrancy [70]. Hang et al. [27] and M-A-R [69] employ dynamic symbolic execution to identify particular vulnerabilities. Mythril uses symbolic execution, SMT solving, and taint analysis to analyze smart contracts on EVM-compatible blockchains [13].

**Fuzzing.** ContractFuzzer is the first fuzzer on Ethereum, presenting several test oracles to detect vulnerabilities [30]. ItyFuzz is a snapshot-based fuzzer that helps reduce analysis overhead [52]. sFuzz utilizes a lightweight multi-objective adaptive strategy to enhance the fuzzing efficiency [45]. SynTest-solidity combines random search and genetic algorithms to generate test cases [46].

**Formal Verification.** Almakhour et al. model the contract as a finite state machine and then perform verification [1]. Solicitous et al. design modeling methodology specifically for Solidity, instead of utilizing pre-existing tools designed for general languages [47].

**Others.** Many studies use techniques such as deep learning [9, 15, 28, 36, 42, 44, 51, 53, 71, 72], knowledge graphs [26], large language models [25, 48], graph neural networks [8, 75] to identify defects.

Tools for Solidity cannot be readily adapted for Move due to the differing instruction sets, runtime environments, syntax, and semantics between the two languages.

## 10 Conclusion

This work presents the first empirical study on Move contracts' security, auditing 652 modules from real-world projects and uncovering eight unique defect types. We introduce MoveScan, an automated analysis framework designed specifically for Move bytecode, demonstrating its superior performance over existing tools. By applying MoveScan to major blockchains using Move, we discovered 97,028 defects, highlighting the significant presence of these issues in deployed contracts.

## Acknowledgments

The authors thank Aptos Labs and Mysten Labs for their invaluable support and the anonymous reviewers for their constructive comments. This work is partially supported by the National Natural Science Foundation of China (62332004), the Sichuan Provincial Natural Science Foundation for Distinguished Young Scholars (2023NSFSC1963), the Basic Strengthening Program (No.2021-JCJQ-JJ-0463), and the Hong Kong RGC Projects (PolyU15224121, PolyU15231223).

## References


[1] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. 2023. A formal verification approach for composite smart contracts security using FSM. *Journal of King Saud University - Computer and Information Sciences* 35, 1 (2023), 70–86. https://doi.org/10.1016/j.jksuci.2022.08.029

[2] Aptflip. 2024. Aptflip | Flip to win. https://aptflip.com/. Accessed on: April 6, 2024.

[3] Aptoscan. 2024. Daily User Transactions. https://aptoscan.com/. Accessed on: February 7, 2024.

[4] argo. 2024. Argo | USDA. https://argo.fi/. Accessed on: April 6, 2024.

[5] BeosinBlockchainSecurity. 2023. Move Lint. https://github.com/BeosinBlockchainSecurity/Move-Lint. Accessed on: September 13, 2023.

[6] BFlyFinance. 2022. FAI. https://github.com/BFlyFinance/FAI. Accessed on: November 13, 2023.

[7] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. 2019. Move: A language with programmable resources. *Libra Assoc* (2019), 1. https://api.semanticscholar.org/CorpusID:201681125

[8] Jie Cai, Bin Li, Jiale Zhang, Xiaobing Sun, and Bing Chen. 2023. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. *Journal of Systems and Software* 195 (2023), 111550. https://doi.org/10.1016/j.jss.2022.111550

[9] Jie Cai, Bin Li, Tao Zhang, Jiale Zhang, and Xiaobing Sun. 2024. Fine-grained smart contract vulnerability detection by heterogeneous code feature learning and automated dataset construction. *Journal of Systems and Software* 209 (2024), 111919. https://doi.org/10.1016/j.jss.2023.111919

[10] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1503–1520. https://doi.org/10.1145/3319535.3345664

[11] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104 arXiv:https://doi.org/10.1177/001316446002000104

[12] CoinGecko. [n. d.]. Cryptocurrency Prices by Market Cap. https://www.coingecko.com/. Accessed on: February 1, 2024.

[13] Consensys. [n. d.]. Mythril. https://github.com/Consensys/mythril. Accessed on: January 30, 2024.

[14] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1-10 (2001), 1–8.

[15] Weichu Deng, Huanchun Wei, Teng Huang, Cong Cao, Yun Peng, and Xuan Hu. 2023. Smart Contract Vulnerability Detection Based on Deep Learning and Multimodal Decision Fusion. *Sensors* 23, 16 (2023). https://doi.org/10.3390/s23167246

[16] ABEL Finance. 2024. Abel Finance - Decentralized LP Lending Platform. https://abelfinance.xyz/. Accessed on: April 6, 2024.

[17] Aptos Foundation. 2023. Aptos Ecosystem. https://github.com/aptos-foundation/ecosystem-projects. Accessed on: November 13, 2023.

[18] Aptos Foundation. 2024. Aptos - The World's Most Production-Ready Blockchain. https://aptosfoundation.org/. Accessed on: January 22, 2024.

[19] Axelar Foundation. 2024. Axelar | Internet-scale interoperability. https://axelar.network/. Accessed on: April 6, 2024.

[20] Aptos Foundation. 2024. Move Specification Language. https://aptos.dev/move/prover/spec-lang/. Accessed on: March 7, 2024.

[21] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 728–739. https://doi.org/10.1145/3533767.3534378

[22] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. (May 2023), 945–956. https://doi.org/10.1109/ICSE48619.2023.00087

[23] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2020. MadMax: analyzing the out-of-gas world of smart contracts. *Commun. ACM* 63, 10 (sep 2020), 87–95. https://doi.org/10.1145/3416262

[24] Wolfgang Grieskamp. 2024. Boogie type error with bit vector number representatioon. https://github.com/aptos-labs/aptos-core/issues/12526. Accessed on: March 25, 2024.

[25] Zheyuan He, Zihao Li, Sen Yang, Ao Qiao, Xiaosong Zhang, Xiapu Luo, and Ting Chen. 2024. Large Language Models for Blockchain Security: A Systematic Literature Review. arXiv:2403.14280 [cs.CR] https://arxiv.org/abs/2403.14280

[26] Tianyuan Hu, Bixin Li, Zhenyu Pan, and Chen Qian. 2024. Detect Defects of Solidity Smart Contract Based on the Knowledge Graph. *IEEE Transactions on Reliability* 73, 1 (March 2024), 186–202. https://doi.org/10.1109/TR.2023.3233999

[27] Jianjun Huang, Jiasheng Jiang, Wei You, and Bin Liang. 2022. Precise Dynamic Symbolic Execution for Nonuniform Data Access in Smart Contracts. *IEEE Trans. Comput.* 71, 7 (July 2022), 1551–1563. https://doi.org/10.1109/TC.2021.3092639

[28] Meng Huang, Jia Yang, and Cong Liu. 2023. CDRF: A Detection Method of Smart Contract Vulnerability Based on Random Forest. In *Provable and Practical Security*, Mingwu Zhang, Man Ho Au, and Yudi Zhang (Eds.). Springer Nature Switzerland, Cham, 407–428.

[29] Chainbase Inc. 2023. Chainbase.com: The #1 Web3 Data Infra. https://chainbase.com/. Accessed on: November 13, 2023.

[30] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[31] Eric Keilty, Keerthi Nelaturu, Bowen Wu, and Andreas Veneris. 2022. A Model-Checking Framework for the Verification of Move Smart Contracts. In *2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS)*. 1–7. https://doi.org/10.1109/ICSESS54813.2022.9930214

[32] Bware Labs. 2024. Decentralized Web3 Infrastructure - Blast API. https://blastapi.io/. Accessed on: April 6, 2024.

[33] Mysten Labs. 2023. Awesome Move. https://github.com/MystenLabs/awesome-move. Accessed on: November 13, 2023.

[34] Zhou Liao, Shuwei Song, Hang Zhu, Xiapu Luo, Zheyuan He, Renkai Jiang, Ting Chen, Jiachi Chen, Tao Zhang, and Xiaosong Zhang. 2023. Large-Scale Empirical Study of Inline Assembly on 7.6 Million Ethereum Smart Contracts. *IEEE Transactions on Software Engineering* 49, 2 (2023), 777–801. https://doi.org/10.1109/TSE.2022.3163614

[35] Yiping Liu, Jie Xu, and Baojiang Cui. 2022. Smart Contract Vulnerability Detection Based on Symbolic Execution Technology. In *Cyber Security*, Wei Lu, Yuqing Zhang, Weiping Wen, Hanbing Yan, and Chao Li (Eds.). Springer Nature Singapore, Singapore, 193–207.

[36] Zhenpeng Liu, Mingxiao Jiang, Shengcong Zhang, Jialiang Zhang, and Yi Liu. 2023. A Smart Contract Vulnerability Detection Mechanism Based on Deep Learning and Expert Rules. *IEEE Access* 11 (2023), 77990–77999. https://doi.org/10.1109/ACCESS.2023.3298048

[37] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[38] MITRE. 2024. CWE-252: Unchecked Return Value. https://cwe.mitre.org/data/definitions/252.html. Accessed on: March 7, 2024.

[39] Move. 2024. Move standard library. https://github.com/move-language/move/tree/main/language/move-stdlib. Accessed on: April 6, 2024.

[40] move language. 2023. Introduction - The Move Book. https://move-language.github.io/move/. Accessed on: January 22, 2024.

[41] MoveBit. 2023. MoveBit: Pioneer in Move Security. https://movebit.xyz/. Accessed on: November 13, 2023.

[42] K. Lakshmi Narayana and K. Sathiyamurthy. 2023. Automation and smart materials in detecting smart contracts vulnerabilities in Blockchain using deep learning. *Materials Today: Proceedings* 81 (2023), 653–659. https://doi.org/10.1016/j.matpr.2021.04.125 International Virtual Conference on Sustainable Materials (IVCSM-2k20).

[43] Keerthi Nelaturu, Eric Keilty, and Andreas Veneris. 2023. Natural Language-Based Model-Checking Framework for Move Smart Contracts. In *2023 Tenth International Conference on Software Defined Systems (SDS)*. 89–94. https://doi.org/10.1109/SDS59856.2023.10328964

[44] Hoang H. Nguyen, Nhat-Minh Nguyen, Chunyao Xie, Zahra Ahmadi, Daniel Kudendo, Thanh-Nam Doan, and Lingxiao Jiang. 2023. MANDO-HGT: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection. (May 2023), 334–346. https://doi.org/10.1109/MSR59073.2023.00052

[45] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. https://doi.org/10.1145/3377811.3380334

[46] Mitchell Olsthoorn, Dimitri Stallenberg, Arie van Deursen, and Annibale Panichella. 2022. SynTest-solidity: automated test case generation and fuzzing for smart contracts. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 202–206. https://doi.org/10.1145/3510454.3516869

[47] Rodrigo Otoni, Matteo Marescotti, Leonardo Alt, Patrick Eugster, Antti Hyvärinen, and Natasha Sharygina. 2023. A Solicitous Approach to Smart Contract Verification. *ACM Trans. Priv. Secur.* 26, 2, Article 15 (mar 2023), 28 pages. https://doi.org/10.1145/3564699

[48] Ferda Özdemir Sönmez and William J. Knottenbelt. 2024. ContractArmor: Attack Surface Generator for Smart Contracts. *Procedia Comput. Sci.* 231, C (apr 2024), 8–15. https://doi.org/10.1016/j.procs.2023.12.151

[49] Junkil Park, Teng Zhang, Wolfgang Grieskamp, Meng Xu, Gerardo Di Giacomo, Kundu Chen, Yi Lu, and Robert Chen. 2024. Securing Aptos Framework with Formal Verification. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024) (Open Access Series in Informatics (OASIcs), Vol. 118)*, Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:16. https://doi.org/10.4230/OASIcs.FMBC.2024.9

[50] Michele Pasqua, Andrea Benini, Filippo Contro, Marco Crosara, Mila Dalla Preda, and Mariano Ceccato. 2023. Enhancing Ethereum smart-contracts static analysis by computing a precise Control-Flow Graph of Ethereum bytecode. *Journal of Systems and Software* 200 (2023), 111653. https://doi.org/10.1016/j.jss.2023.111653

[51] Christoph Sendner, Huili Chen, Hossein Fereidooni, Lukas Petzi, Jan König, Jasper Stang, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2023. Smarter Contracts: Detecting Vulnerabilities in Smart Contracts with Deep Transfer Learning. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/smarter-contracts-detecting-vulnerabilities-in-smart-contracts-with-deep-transfer-learning/

[52] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 322–333. https://doi.org/10.1145/3597926.3598059

[53] Shuxiao Song, Xiao Yu, Yuexuan Ma, Jiale Li, and Jie Yu. 2024. Multi-model Smart Contract Vulnerability Detection Based on BiGRU. In *Neural Information Processing*, Biao Luo, Long Cheng, Zheng-Guang Wu, Hongyi Li, and Chaojie Li (Eds.). Springer Nature Singapore, Singapore, 3–14.

[54] Donna Spencer and Todd Warfel. 2004. Card sorting: a definitive guide. *Boxes and arrows* 2, 2004 (2004), 1–23.

[55] Staketab. 2023. Suiscan: Sui Explorer. https://suiscan.xyz/mainnet/home. Accessed on: November 13, 2023.

[56] Starcoin. 2023. Starcoin Framework. https://github.com/starcoinorg/starcoin-framework. Accessed on: November 13, 2023.

[57] Starcoin. 2024. Transactions - Starcoin. https://stcscan.io/main/transactions/1. Accessed on: February 7, 2024.

[58] Starcoin.org. 2021. Starcoin: Homepage. https://starcoin.org/en/. Accessed on: January 22, 2024.

[59] Elements Studio. 2022. Poly Stc contracts. https://github.com/Elements-Studio/poly-stc-contracts. Accessed on: November 13, 2023.

[60] Elements Studio. 2022. Starswap core. https://github.com/Elements-Studio/starswap-core. Accessed on: November 13, 2023.

[61] Sui. 2023. Sui | Unlock the freedom to build powerful on-chain assets. https://sui.io/. Accessed on: January 22, 2024.

[62] Sui. 2024. Capability - Move by Example. https://examples.sui.io/patterns/capability.html. Accessed on: April 6, 2024.

[63] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain* (Gothenburg, Sweden) *(WETSEB '18)*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/3194113.3194115

[64] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7, Article 148 (jul 2021), 38 pages. https://doi.org/10.1145/3464421

[65] Anna Vacca, Andrea Di Sorbo, Corrado A. Visaggio, and Gerardo Canfora. 2021. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *Journal of Systems and Software* 174 (2021), 110891. https://doi.org/10.1016/j.jss.2020.110891

[66] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug Characteristics in Blockchain Systems: A Large-Scale Empirical Study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 413–424. https://doi.org/10.1109/MSR.2017.59

[67] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart Contract Security: A Practitioners' Perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1410–1422. https://doi.org/10.1109/ICSE43902.2021.00127

[68] Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li, Rui Qin, and Fei-Yue Wang. 2018. An Overview of Smart Contract: Architecture, Applications, and Future Trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. 108–113. https://doi.org/10.1109/IVS.2018.8500488

[69] Zexu Wang, Bin Wen, Ziqiang Luo, and Shaojie Liu. 2021. M-A-R: A Dynamic Symbol Execution Detection Method for Smart Contract Reentry Vulnerability. In *Blockchain and Trustworthy Systems*, Hong-Ning Dai, Xuanzhe Liu, Daniel Xiapu Luo, Jiang Xiao, and Xiangping Chen (Eds.). Springer Singapore, Singapore, 418–429.

[70] Rutao Yu, Jiangang Shu, Dekai Yan, and Xiaohua Jia. 2021. ReDetect: Reentrancy Vulnerability Detection in Smart Contracts with High Accuracy. In *2021 17th International Conference on Mobility, Sensing and Networking (MSN)*. 412–419. https://doi.org/10.1109/MSN53354.2021.00069

[71] Dawei Yuan, Xiaohui Wang, Yao Li, and Tao Zhang. 2023. Optimizing smart contract vulnerability detection via multi-modality code and entropy embedding. *Journal of Systems and Software* 202 (2023), 111699. https://doi.org/10.1016/j.jss.2023.111699

[72] Hengyan Zhang, Weizhe Zhang, Yuming Feng, and Yang Liu. 2023. SVScanner: Detecting smart contract vulnerabilities via deep semantic extraction. *Journal of Information Security and Applications* 75 (2023), 103484. https://doi.org/10.1016/j.jisa.2023.103484

[73] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L. Dill. 2020. The Move Prover. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 137–150.

[74] Andrei Zhukov and Vladimir Korkhov. 2023. SmartGraph: Static Analysis Tool for Solidity Smart Contracts. In *Computational Science and Its Applications – ICCSA 2023 Workshops*, Osvaldo Gervasi, Beniamino Murgante, Ana Maria A. C. Rocha, Chiara Garau, Francesco Scorza, Yeliz Karaca, and Carmelo M. Torre (Eds.). Springer Nature Switzerland, Cham, 584–598.

[75] Lihan Zou, Changhao Gong, Zhen Wu, Jie Tan, Junnan Tang, Zigui Jiang, and Dan Li. 2024. A General Smart Contract Vulnerability Detection Framework with Self-attention Graph Pooling. In *Blockchain and Trustworthy Systems*, Jiachi Chen, Bin Wen, and Ting Chen (Eds.). Springer Nature Singapore, Singapore, 3–16.

Received 2024-04-12; accepted 2024-07-03