JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 38, 1171-1188 (2022)
DOI: 10.6688/JISE.202211_38(6).0005

A Novel Detection Method for the Security Vulnerability of Time-of-Check to Time-of-Use*

YUNGYU ZHUANG+ AND YAO-NANG TSENG
Department of Computer Science and Information Engineering
National Central University
Taoyuan, 32001 Taiwan
E-mail: yungyu@ncu.edu.tw; 106522031@cc.ncu.edu.tw

Since Artificial Intelligence (AI) is applied to various applications for intelligent and automatic processing, ensuring systems security is even important. Many developers still prefer C-like languages for flexibility, usability, and historical reasons to implement underlay systems, though other languages support more modern features. As a result of lacking higher-level abstraction and exception handling, languages like C are known to risk several security vulnerabilities. Time-of-Check to Time-of-Use (TOCTOU) is one of the security vulnerabilities in C codes, a kind of bug caused by race conditions. Unexpected use of certain function calls might be executed and result in failure or abnormal behaviors of systems if someone injects malicious operations between the time of check on system status and the use of the check result. Several research activities on code analysis, including static and dynamic approaches, were devoted to developing detection methods, but there is room for improvement. We propose a novel method to statically detect the TOCTOU vulnerability and implement a tool built atop of a solid static analyzer to show the feasibility of our idea. Our tool was evaluated with the test cases for TOCTOU vulnerabilities and compared with existing detection methods. The results show that our method can detect TOCTOU vulnerabilities more accurately and cover all possible paths in the source code.

Keywords: security vulnerability, source code analysis, static analysis, time-of-check to time-of-use, TOCTOU

1. INTRODUCTION

Even today C language is still one of the most popular programming languages [1]. A lot of software projects rely on its flexibility and usability. Although nowadays there are many new programming languages with modern features such as lambda functions, classes/objects, and generics, C language is still widely used in some domains, including operating systems and embedded systems. The reason why many developers prefer the C language might include the performance advantage over other modern languages and the existence of legacy code. Since C language is relatively low-level, programs written in C language are expected to be run faster. Furthermore, it is supported by a lot of compilers with dedicated optimizations. For example, usually there are only a few languages supported on customized hardware to benefit from special instructions, and C language is always one of them. Consequently, being a relatively low-level language, programs written in C language tend to have security vulnerabilities due to the lack of modern abstraction and well-defined resulting behaviors [2]. Although experienced programmers can avoid

Received October 13, 2021; revised November 15, 2021; accepted November 23, 2021.
Communicated by Shin-Jie Lee.
* This work was supported in part by the Ministry of Science and Technology under Grand No. MOST 107-2221-E-008-024-MY3.

falling into such traps, getting this know-how for beginners is not easy. Moreover, there are a lot of legacy programs written in the C language, and no one can assume that every piece of code was written by experienced programmers.

Time-of-Check to Time-of-Use (TOCTOU) is a known security vulnerability that might lead to serious system problems [3]. Dean and Hu [4] have shown that there is no portable and deterministic solution without changes to the system call interface. The TOCTOU vulnerability is a kind of bug due to race conditions, which may occur unexpectedly in certain cases. A race condition is the condition of a system where two or more threads access the same variables or objects concurrently and at least one thread performs updates. The system might behave differently every time since the execution order depends on how these threads are scheduled. Thus, race conditions can be considered as a vulnerability related to thread scheduling. For example, Fig. 1 shows two threads that access the same variable x in a program. Thread A will set the value of x to 10 first, do some operations, and then get the value of x for setting y. On the other hand, during the execution of Thread A, Thread B will set the value of x to 20. In the case of Fig. 1 (a), Thread A gets 10 for the variable x since it executes "y = x" before Thread B executes "x = 20". Since these threads are run based on thread scheduling and there is no constraint on accessing x, the thread execution might look like Fig. 1 (b) next time. In this case, Thread A gets 20 for the variable x, which might be undesirable. Without any variable access control, we cannot ensure what Thread A will get, and thus the program's behavior is uncontrollable. Race conditions are also discussed in various domains such as electronics and networking, but here we focus on systems data access.

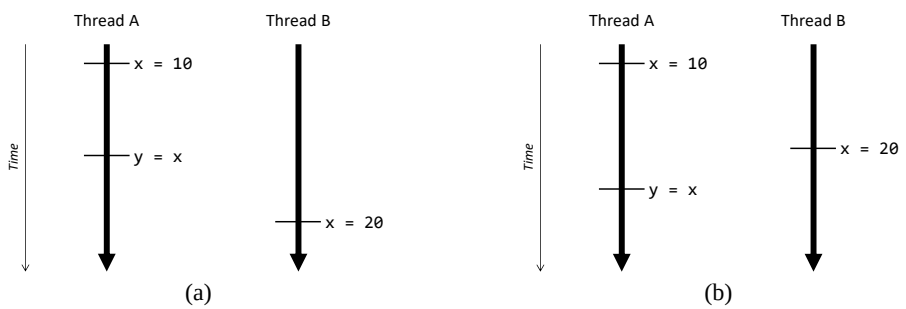


Fig. 1. An example of race conditions, where Thread A might get different values for x every time.

The root cause of TOCTOU is the temporal difference between the check on a system state and the use of the check result. In a system that has such a vulnerability, someone can attack the system with another piece of code that performs something between the check and the use of this check result. The problem is similar to the one shown in Fig. 1, but now A and B are processes running on a system and x represents a system state as shown in Fig. 2. Process A will check the system state x and perform certain operations later based on the check result. However, by executing a malicious code along with the code containing the check and its use, there is a possibility that an unexpected modification will be performed between the check and its use. For example, Process A might normally work as shown in Fig. 2 (a) or behavior unexpectedly based on the changed x as shown in Fig. 2 (b). In other words, the users of a system can try to execute malicious code that changes

something just after the check to make the use of this check result no longer safe. It means the actions that rely on this check result might not work as expected since the system's state has been changed since the check. The TOCTOU vulnerability is a typical issue on file access. For example, malicious users may replace the file to use with a symbolic link to another vital file after the permission check of the file; the action expected to be performed on the file will now be performed on that vital file. How to avoid the TOCTOU attack can be considered at the system level. Several research activities worked on different scenarios, such as improving system architecture design to prevent illegal usage or implementing a mechanism to trap malicious operations [5-7].

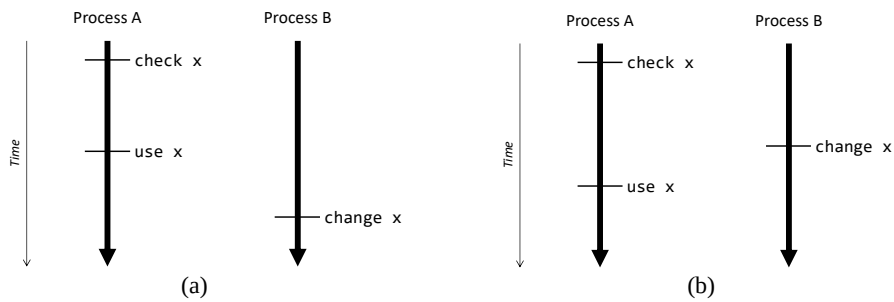


Fig. 2. Process B is executing a malicious code that tries to change the system state x.

The TOCTOU vulnerability can also be considered and prevented at the language level [8], but it is not the case with C language. Suppose that an exception handling mechanism is directly given as language constructs, for example the exceptions in Java and Python. Then programmers can do certain actions without any check and leaving problems to error handling – the failure of assumption will be detected at the time of use. Unfortunately, not every language gives constructs for exception handling, for example C language. Without the help of exception handling, programmers might write a code containing time-of-check and time-of-use, which may result in the TOCTOU vulnerability. Note that the TOCTOU vulnerability is not an issue that only happens in C language but is often discussed in C language due to its popularity in system implementation. In this case, an effective way to avoid TOCTOU vulnerabilities is analyzing programs by tools at the code level in advance, *i.e.*, using a code analysis tool to check the source code of programs and trying to fix them before deploying these programs.

The ways to detect such a vulnerability can be briefly categorized into two kinds of approaches: the dynamic analysis approach and the static analysis approach. Dynamic analysis [9-13] is to analyze programs by executing them directly. The analyzers in this category usually need to insert instructions into object programs before running them in order to gather necessary information and know the states of programs at runtime. Since this approach can explore only a single execution path every time, it has difficulty in ensuring the coverage of analysis practically [14, 15]. On the other hand, the static analysis approach [16-23] analyzes programs without really executing them. Static analyzers are a tool that reads source code, analyzes the code, and generates a report on the analysis results. The detection methods used in this approach include pattern matching, lexical analysis, and parsing analysis. Pattern matching can quickly search and match a specific pattern in

a given code, but it is unaware of language semantics. For example, comments and macros cannot be recognized well as a result of handling source code as strings. On the other hand, lexical analysis and parsing analysis are based on compiler architecture. Lexical analysis further uses the lexical information inside compilers, making it aware of tokens without syntactic meaning. Parsing analysis can understand syntactic meaning by constructing abstract syntax trees. These methods are basically more expensive but more powerful since they have the knowledge of language semantics. In general, the code coverage of static analysis is better than dynamic analysis since all execution paths in the source code will be considered. However, the source code of programs to detect must be provided.

In this paper, we propose a novel method to detect the TOCTOU vulnerability in C-like language based on static analysis with parsing techniques. This proposal aims to find out all possibilities of the TOCTOU vulnerability in programs while eliminating false-positive results as many as possible – to avoid incorrectly recognizing vulnerabilities. Although our method is not limited to C language and can be implemented for similar languages, we give a concrete implementation for C language to show the feasibility and usability of our proposal. We evaluate the accuracy of our tool with Juliet Test Suite [24, 25] and compare the ability of our tool with several related works. In addition to comparing with six non-commercial tools for general vulnerabilities by running our test cases, we also list features to compare with six existing research activities on TOCTOU detection.

2. RELATED WORK

Several research activities are devoted to the TOCTOU vulnerability, including those working in file access [3, 26-30] and trusted computing [5-7, 31, 32]. They are targeted at either finding the TOCTOU vulnerability in source code or monitoring the system to prevent TOCTOU attacks. This section explains the techniques used in our proposal and summarizes related works based on them.

2.1 System Call Pairing

The analysis tool proposed by Bishop and Dilger [3] uses pattern matching over the given C code to generate a call dependency. Based on the pairing knowledge on system calls, the tool can determine potential programming intervals, *i.e.*, the interval between the check and the use of check results. For example, by assuming the pairing relation between the system calls open and access, it can detect the existence of TOCTOU vulnerability:

```
if (access("file", W_OK) != 0) {
            :
fd = open("file", O_WRONLY);
```

where access and open are included in the set of check function and use function, respectively. Note that the interval between system calls is temporal rather than spatial. As a consequence of adopting pattern matching, the tool is lightweight and fast but does not have the knowledge of language semantics. Since no data flow analysis is performed, the tool has no idea about inter-procedural analysis and pointer aliasing. Viega *et al.* proposed a static vulnerability scanner named ITS4 for C and C++ code based on lexical analysis

[26]. Since ITS4 performs real parsing, it can avoid a false positive in the case of declaring a variable with the name of system calls. For example, the following declaration will not trigger the detection:

```
int access;
```

since the identifier access is a variable name rather than a function call. However, the aliasing problem was not addressed by ITS4 and listed as a future direction.

2.2 Symbolic Execution

Symbolic execution [33-37] is a promising technique to improve static code analysis. It converts variables and values with abstract symbols and emulates the execution of programs by handling these abstract symbols to represent the states of programs. During the emulation, it needs to fork and maintain states for conditional branches. Fig. 3 shows an example of conditional branches, where the function call trigger is the place that triggers a bug, *i.e.*, the bug occurs when 40 <= x < 100. In symbolic execution, the states of this piece of code will be emulated, as shown in Fig. 4. For every if-else branch, it will fork two states and fill in the conditions for x in the states. Eventually, the state we would like to find is the one named AAA. In some sense, symbolic execution can take the states of programs at runtime into account and explore all execution paths in programs based on static code analysis. The TOCTOU detection method proposed by Lai is based on symbolic execution to significantly reduce false-positive results, *i.e.*, reporting the code without the vulnerability. However, although it can accurately determine the execution paths that might have the TOCTOU vulnerability, system calls with different parameters will still be paired and reported as positive.

```
x = getInput();
if (x >= 40) {
    if (x < 100)
        trigger();
    else
        printf("x is so big");
}
else {
    printf("x is so small");
}
```

Fig. 3. Suppose trigger() will cause a bug.

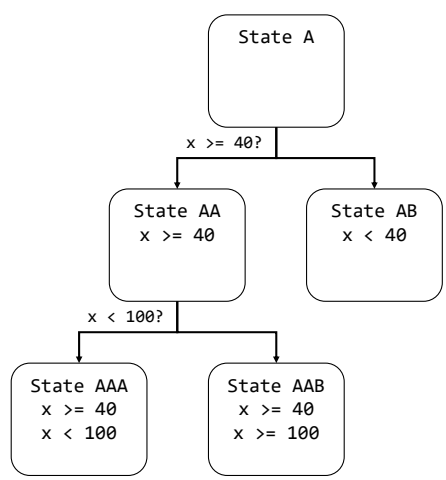


Fig. 4. Using symbolic execution to analyze Fig. 3.

2.3 Parameter Tracking

Parameter tracking, *i.e.* inter-procedural value flow analysis, is a technique used in

several code analysis research activities, including the static ones [3, 26] and the dynamic ones [27-29]. It means tracking the parameters given the function calls and using them to determine the relation between the function calls. For example, the following code seems to contain a TOCTOU vulnerability but actually not:

```
if (access(file1, W_OK) != 0) {
                :
fd = open(file2, O_WRONLY);
```

Here both file1 and file2 are character pointers. In this case, the system calls access and open are possibly different, but tools without parameter tracking will leave it as a false positive. The tools proposed by Bishop and Dilger [3] and Viega *et al.* [26] can find system call pairs according to the parameters, but their recognition ability is limited to pattern matching and lexical analysis, respectively. Without knowing language semantics, it is impossible to understand the relations among variables in a static checker. On the other hand, dynamic analysis tools can monitor system calls at the system level and obtain the actual parameters given at runtime for pairing. Cowan *et al.* proposed a tool named RaceGuard to detect attempts to exploit temporary file race vulnerabilities [27]. RaceGuard is a kernel enhancement and can effectively detect the TOCTOU vulnerability. The kernel monitoring tools built by Wei and Pu also benefit from parameter tracking by recording the parameters given at runtime, particularly filenames [28]. It also clearly defines the sets of check functions and use functions to enumerate TOCTOU pairs in Linux. As successive research, Pu and Wei further developed a defense mechanism [29]. As a systematic design and implementation in the kernel, it works without changing application code or API. On the contrary, it introduces locks and thus has remaining issues, for example, the question of dead-lock and live-lock.

3. MOTIVATION

Time-of-Check to Time-of-Use (TOCTOU) is a well-known vulnerability caused by race conditions. For example, the code in Fig. 5 is a typical example that is executed with setuid, which means raising the permission of execution to root temporarily. It is a general way to temporarily elevate users' privileges of running programs to perform a specific task in the system. However, in some cases, we still need to check whether or not the user executing the program has really been granted permission to a specific file before using the file; otherwise, this program might perform illegal actions. This example looks safe since the program will open and write the specified file only when the user executing the program has the permission checked by the access function call. However, a malicious user might let the code in Fig. 6 be executed on the same system, and the timing of the execution may be just between the access function call and the open function call. For example, suppose the remove and symlink function calls in Fig. 6 are executed between the calls to access and open in Fig. 5. In that case, the malicious program will immediately remove the file after the check (access function call) and replace it with a symbolic link to the critical system file passwd. Thus, after the calls to open and write, the system file will be modified unexpectedly. In other words, although the check ensures the state is correct at that moment, the check result might be no longer reliable when we really want to use it.

```
if (access("file", W_OK) != 0) {
    exit(1);
}
fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

Fig. 5. A typical example executed with setuid.

```
// after access()
remove("file");
symlink("/etc/passwd", "file");
// before open()
```

Fig. 6. A malicious code targeted at Fig. 5.

A general solution to avoid leaving the space of time is applying exception handling, which tries to do actions directly and handles failure cases without knowing the state in advance. Instead of checking the state before doing the action, we can have assumptions on the program's state, do the action directly, and prepare the next steps for different results of doing the action. Thus, exception handling eliminates the space of time between the check and the use. However, in several cases, it is challenging to apply exception handling to the code simply. First, in a language without the direct support for exception handling, for example C language, it is very challenging to implement such an exception handling mechanism by ourselves. Second, there might be a lot of legacy code pieces written in C-like languages, and manually tracing and rewriting all the codes does not make sense.

Therefore, an accurate detection tool for the TOCTOU vulnerability is helpful. If we can analyze the code and accurately detect TOCTOU vulnerabilities in legacy code, we may focus on where the vulnerability exists and fix it effectively. The ways to analyze code can be generally classified into two categories, as mentioned in Section 1: static analysis and dynamic analysis. Although static analysis lacks runtime information in program execution, it can cover all execution paths in a program and emulate the execution with the help of symbolic execution. On the other hand, parameter tracking is usually used in dynamic analysis, which can help to understand the relations of parameters to functions. So far as we know, no existing tool is applying this technique with symbolic execution for detecting the TOCTOU vulnerability.

The observation on the necessity of TOCTOU detection and the usability of symbolic execution and parameter tracking led us to dive into the research of developing a TOCTOU detection tool based on static code analysis. By studying related works [3, 26-30], we developed a detection method that can find out TOCTOU vulnerabilities more accurately with fewer false-positive results – the ones that are not vulnerabilities but are being recognized as vulnerabilities.

4. OUR DESIGN AND IMPLEMENTATION

We propose a detection method for the TOCTOU security vulnerability based on static code analysis, which combines symbolic execution and parameter tracking. The method proposed by Lai [30] adopts symbolic execution to cover all the execution paths in a program. Although the introduction of symbolic execution is not cheap, it can significantly improve the correctness of detection. However, due to the lack of parameter tracking, it might report the code with no vulnerability, *i.e.*, false-positive results. On the other hand, parameter tracking is an approach to find out the relation between check functions and use functions by matching their parameters, such as the filename given to these functions. Several proposed methods [3, 26] use parameter tracking on static code analysis, but

these methods do not benefit from the syntactic meaning of code. Other proposed methods [27-29] use parameter tracking on dynamic code analysis, but they cannot find the vulnerability in all execution paths due to the nature of dynamic code analysis.

4.1 A Quick Overview

Our proposal is based on parsing analysis and follows the idea used by Wei and Pu [28] to separate functions into two groups for pairing system calls. Furthermore, it integrates symbolic execution with parameter tracking for reducing false positives as much as possible. By enumerating functions in the check group and the use group, we can explicitly manage and pair system calls. In our current design, the set structure is used for grouping, and a system call can be listed in both groups. Since pairing the system calls appearing in different execution paths is not reasonable, we apply symbolic execution to consider all execution paths and exclude such unreasonable cases. For example, placing the calls to a check function and a use function in different blocks of an if-else branch is actually a safe case and should be ignored. For risky cases, we can report the existence of a TOCTOU vulnerability along with the conditions resulting in the execution path. For example, if a use function is placed inside a nested if-else block, the value range of variables used in the if-clauses will be reported as a clue to examine the risk. In order to further reduce the number of incorrect reports, the parameters are tracked during the pairing of check functions and use functions. System calls on different targets will not be paired with the help of reasoning about the semantics of code.

4.2 The Design of Our Detection Method

We prepare two sets, CheckSet and UseSet, for maintaining the two groups of functions: check functions and use functions, respectively. Currently, we only added Linux system calls in our tool implementation as follows, but it is possible to add library function calls as much as we need:

```
CheckSet = {access, stat, open, creat, mknod, link, symlink, mkdir, unlink, rmdir, rename,
            execve, chmod, chown, truncate, utime, chdir, chroot, pivot_root, mount}

UseSet = {creat, mknod, mkdir, rename, link, symlink, open, execve, chdir, chroot,
          pivot_root, mount, chmod, chown, truncate, utime}
```

Note that functions might be included in both CheckSet and UseSet, depending on how they are used in programs. In addition to the two sets, our tool uses a map to track the parameters of functions:

```
Map<Var, VarState>
```

Here Var is a variable that might be used as parameters of functions, and VarState is a data structure containing the following information:

```
(Status, CheckFunction, UseFunction)
```

where Status is the current status of this variable and can be set to Unchecked, Checked, or Used. CheckFunction and UseFunction are used to store the names of the functions in the CheckSet and UseSet that use the variable as a filename parameter, respectively. This map records the variables that are used in parameters given to functions and maintains their

status. The status of a variable will be updated when our tool finds the functions in CheckSet and UseSet as shown in Fig. 7. It is updated as follows:

1. When Status is Unchecked and our tool finds a function in CheckSet, setting CheckFunction to the name of the function and changing Status to Checked. It marks the occurrence of time of check. If later we find an occurrence of time of use on the same variable, it will become a TOCTOU vulnerability.
2. When Status is Checked and the tool finds another function in CheckSet, updating CheckFunction to the name of this function without changing Status. The transmission is to remember the latest occurrence of time of check.
3. When Status is Checked and the tool finds a function in UseSet, setting UseFunction to the name of the function and changing Status to Used. It notes that a TOCTOU vulnerability arises.
4. When Status is Used, pushing the vulnerability information, *i.e.* the pair of CheckFunction and UseFunction, into the stack for reporting vulnerabilities, then clearing CheckFunction and UseFunction and setting Status to Unchecked for the next pairing.

The status transmission is updated per variable, and it ensures the latest check function call can be paired with the first use function call. Staying at the status of Unchecked or Checked represents a safe case while reaching the status of Used suggests a risky case, *i.e.*, the existence of a TOCTOU vulnerability.

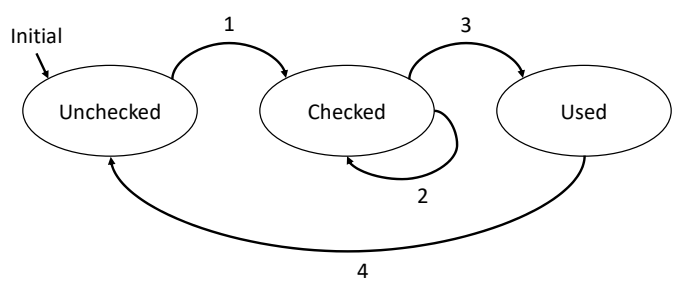


Fig. 7. The status transmission of a variable in our detection method.

When forking the states for a conditional branch in symbolic execution, the status of variables will also be forked – every forked state inherits the status from its parent. In other words, we want to find TOCTOU vulnerabilities in all conditional branches by maintaining the status transmission in them. Taking Fig. 8 as an example, which is modified from the code in Fig. 3, the states of analyzing it with symbolic execution will look like Fig. 9. In the beginning, `x` might be any value, and Status is Unchecked. After the first conditional branch "`x >= 40?`", the state will be forked: one for "`x >= 40`" and one for "`x < 40`", and their statuses are initialized as Unchecked by inheriting from their parent. At this moment, both two cases are safe since no check function was detected. In the execution path of "`x >= 40`", Status is set to Checked since a CheckFunction named `access` is found. It marks the beginning of pairing, but it is still safe since no use function has been found yet. After the second conditional branch, "`x < 100?`", the state will be further forked: one for "`x < 100`" and one for "`x >= 100`", and their statuses are initialized as Checked as their parent. In the

path of "x < 100", Status will be set to Used to report the existence of a TOCTOU vulnerability since a UseFunction named open is called here. In the example, only the execution paths represented by the state AAA is a risky case, *i.e.*, there is a TOCTOU vulnerability, and the remainders are safe cases.

```
x = getInput();
if (x >= 40) {
    access("file", W_OK);
    if (x < 100)
        open("file", O_WRONLY);
    else
        printf("x is so big");
}
else {
    printf("x is so small");
}
```

Fig. 8. An example of TOCTOU vulnerability.

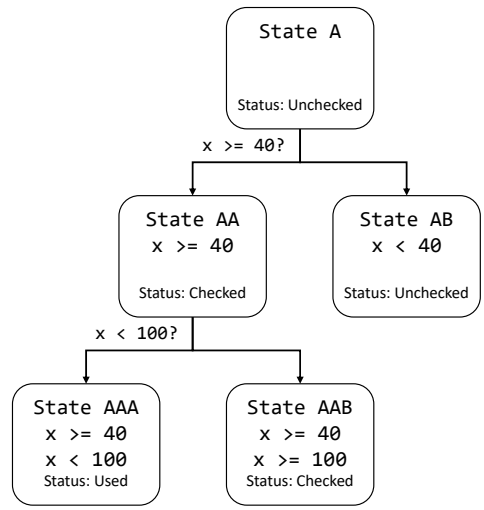


Fig. 9. The states of analyzing Fig. 8.

The pairing of CheckFunction and UseFunction in our detection method is based on the variables they use rather than simply pairing them. We track the variables used as parameters in functions like access and open to avoid trapping the cases that are not really a TOCTOU vulnerability – the one to use is actually different from the one to check. For example, in Fig. 8, if the parameter used in the open function call is "file2" rather than "file", the status of VarState for "file" will not be set to Used, and thus no vulnerability will be reported. Note that we focused on file names in parameters in our current implementation since file access is the major issue in the TOCTOU vulnerability. By applying parameter tracking with symbolic execution, our tool can detect the TOCTOU vulnerability more accurately since the false-positive results can be significantly reduced.

4.3 The Implementation Details

We implemented our tool[1] on the top of Clang Static Analyzer (CSA), a source code analysis tool built on top of Clang and LLVM[2]. The LLVM project is a modular and reusable compiler infrastructure, which supports multiple languages on various platforms. The frontends for different languages translate source code to LLVM IR, and the backends are responsible for generating code for different platforms based on the LLVM IR. Developers may extend the supported languages or platforms by implementing corresponding front-

[1] The implementation of our proposal and the test cases we evaluated are available on the project page: https://github.com/ncu-psl/TOCTOU-Detection.
[2] The version of LLVM/Clang that our tool uses is 3.4.

ends/backends, respectively. CSA can build a memory model for static analysis of C programs and perform symbolic execution [38]. We based our implementation on CSA to benefit from Clang and LLVM.

In our implementation, we maintain two kinds of map structures for the one explained in the previous section:

```
map<std::string, VarState>
map<clang::ento::SymExpr*, VarState>
```

The former is to store the name of variables used in parameters with string literals, and the latter is to store the ones with pointers to objects. For the VarState in the two maps, we used the class shown in Fig. 10 to represent the data structure and wrap up related operations since CSA is implemented in C++. We need one more map in addition to the one for string literals because CSA uses object pointers to analyze assignments. When CSA emulates an assignment, the object pointed by the left-hand side will be set to the one pointed by the right-hand side. It means that we can recognize aliases by holding and comparing these object pointers in assignments. Recognizing aliases enables the ability to pair CheckFunction and UseFunction more correctly. If a variable alias to the string "file", for example "filestr", is given to the open function call instead of "file" in Fig. 8, our detection method can also correctly pair open and access. Similarly, if a function alias to open function named fp is used in the place of open, it can be recognized as a TOCTOU vulnerability as well. This ability is quite crucial since aliases and function pointers are supported and frequently used in C language.

```
class VarState {
private:
    enum Status {Unchecked, Checked, Used} S;
    string CheckFunction, UseFunction;
        :
}
```

Fig. 10. The class used to represent the status and pairing in our proposal.

Thanks to the modular design of LLVM and the power of CSA, we can implement our proposal as a tool that is registered as a callback function. Whenever the analyzer finds a function call, our tool will be invoked. CSA can build data structure from source code and perform static code analysis with symbolic execution. Furthermore, our tool benefits from the built-in tools, scan-build and scan-view, to analyze object programs during compilation and to generate HTML files for reading analysis reports on Web browsers.

5. EVALUATION AND DISCUSSION

To evaluate our proposal, we conducted three kinds of experiments and comparisons. We first run with the test cases inside Juliet Test Suite to show the essential ability to detect TOCTOU vulnerabilities. Then we created additional test cases to compare with six non-commercial tools developed for general vulnerabilities to highlight the features in our tool.

Finally, to compare with existing research activities on TOCTOU detection, we further used several metrics to determine the abilities and limitations of these detection methods. The results show that our tool can accurately detect these test cases and our proposal has advantages over existing detection methods.

5.1 Running with Juliet Test Suite

We run the test cases for the TOCTOU vulnerability inside Juliet Test Suite v1.3 [24, 25] to see whether our tool can accurately detect these typical TOCTOU vulnerabilities or not. Juliet Test Suite is a collection of test cases in the C/C++ language and contains examples organized under 118 common weakness enumerations (CWEs). It was created by the NSA's Center for Assured Software (CAS) for use in testing static analysis tools. For the category of TOCTOU, it gives 100 test cases including 36 test cases that have TOCTOU vulnerabilities.

We evaluated our tool by analyzing these 100 test cases for the TOCTOU vulnerability. The results of detecting the 100 test cases are shown in Table 1. The columns "Positive" and "Negative" under "Actual" mean whether the test cases really have TOCTOU vulnerabilities or not, respectively, and the rows for "Detect" mark the detection results of our tool. The results show that our tool has neither false positive nor false negative. Our tool can correctly distinguish between positive and negative; it reported positive for all positive test cases and reported negative for all negative ones. However, it is hard to say the test cases for the TOCTOU vulnerability in Juliet Test Suite are challenging. The results only show that our tool fulfilled the minimal requirements of TOCTOU detection ability.

Table 1. The results of detecting TOCTOU vulnerabilities in Juliet Test Suite.

Total: 100		Actual	
		Positive	Negative
Detect	Positive	True Positive: 36	False Positive: 0
	Negative	False Negative: 0	True Negative: 64

5.2 Comparing with Non-Commercial Tools

In order to further highlight the TOCTOU detection ability of our tool, we manually created four types of test cases based on the following features of our tools along with a trivial case: call order, parameter tracking, variable alias, and function alias. Call order notes considering the order of function calls. Parameter tracking means the necessity of tracking the parameters in functions. Variable alias and function alias are to recognize the aliases to variables and functions, respectively. We prepared two test cases for each type – one for positive and one for negative as shown in Fig. 11. We picked up the following six non-commercial tools for general vulnerabilities from the list in the study conducted by Fatima *et al.* [39]: FlawFinder (version 2.0.8), VCG (version 2.1.0), CppCheck (version 1.82), Splint (version 3.1.2), SPARSE (version 0.5.1), and RATS (version 2.4).

Trivial Case (Positive)

```
1   int f;
2   void foo(char *file){
3
4       if(!access(file, W_OK)){
5           f = open(file, O_RDWR);
6       }
7   }
```

Trivial Case (Negative)

```
1   int f;
2   void foo(char *file){
3       f = open(file, O_RDWR);
4   }
```

Call Order (Positive)

```
1   int f;
2   void bar(char *file){
3       f = open(file, O_RDWR);
4   }
5
6   void foo(char *file){
7
8       if(!access(file, W_OK)){
9           bar(file);
10      }
11  }
```

Call Order (Negative)

```
1   int f;
2   void bar(char *file){
3       f = access(file, W_OK);
4   }
5
6   void foo(char *file){
7       f = open(file, O_RDWR);
8       bar(file);
9   }
```

Parameter Tracking (Positive)

```
1   int f;
2   void foo(char *file1, char *file2){
3
4       int var = access(file1, W_OK)
5       f = open(file1, O_RDWR);
6   }
```

Parameter Tracking (Negative)

```
1   int f;
2   void foo(char *file1, char *file2){
3
4       int var = access(file1, W_OK)
5       f = open(file2, O_RDWR);
6   }
```

Variable Alias (Positive)

```
1   int f;
2   void foo(char *file1){
3
4       char *file2 = file1;
5
6       if(!access(file1, W_OK)){
7           f = open(file2, O_RDWR);
8       }
9   }
```

Variable Alias (Negative)

```
1   int f;
2   //file1 =/= dummy
3   void foo(char *file1, char *dummy){
4
5       char *file2 = file1;
6       char *file3 = dummy;
7
8       if(!access(file1, W_OK)){
9           f = open(file3, O_RDWR);
10      }
11  }
```

Function Alias (Positive)

```
1   int f;
2   void foo(char *file){
3
4       int (*fp)(const char*, int);
5       fp = open;
6
7       if(!access(file, W_OK)){
8           f = fp(file, O_RDWR);
9       }
10  }
```

Function Alias (Negative)

```
1   int f;
2   int dummy(const char*, int){return 0};
3
4   void foo(char *file){
5
6       int (*fp1)(const char*, int);
7       int (*fp2)(const char*, int);
8
9       fp1 = open;
10      fp2 = dummy;
11
12      if(!access(file, W_OK)){
13          f = fp2(file, O_RDWR);
14      }
15  }
```

Fig. 11. The test cases we created for highlighting the four features of our tool.

We compared our tool with the six tools by these test cases, and the results show that our tool can correctly distinguish between positive and negative. The detecting results are shown in Table 2. The columns "P" and "N" under every type of test case note how each tool recognizes the given test case: "TP" means the tool successfully recognized the vulnerability in the test case, "FP" means the tool recognized a vulnerability that actually does not exist, "TN" means the tool did not find any vulnerability and indeed there is no vulnerability, and "FN" means the tool did not find the vulnerability in the test case. The results show that FlawFinder always reports positive, while VCG, CppCheck, Splint, and SPARSE always ignore the possibility of the TOCTOU vulnerability. RATS worked well for the trivial case and the parameter tracking case while failing to recognize in the cases of call order and aliases. On the other hand, our tool passed all the test cases. We summarized the ability of these tools in Table 3, which shows that most tools failed to pass these test cases since they cannot correctly distinguish between positive and negative of the TOCTOU vulnerability. It is not surprising that our tool can work better since these test cases are created based on the issues we address and the other tools might focus on different scenarios or different kinds of vulnerabilities. However, the comparison shows that the detection of the TOCTOU vulnerability is not supported by some mature tools such VCG, CppCheck, Splint, and SPARSE, and tools like FlawFinder and RATS cannot resolve the problems in detecting the TOCTOU vulnerability we address.

Table 2. The detecting results of our tool and the six non-commercial tools.

Test Case / Tool Name	Trivial Case		Call Order		Parameter Tracking		Variable Alias		Function Alias	
	P	N	P	N	P	N	P	N	P	N
Flawfinder	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
VCG	FN	TN	FN	TN	FN	TN	FN	TN	FN	TN
Cppcheck	FN	TN	FN	TN	FN	TN	FN	TN	FN	TN
Splint	FN	TN	FN	TN	FN	TN	FN	TN	FN	TN
Sparse	FN	TN	FN	TN	FN	TN	FN	TN	FN	TN
RATS	TP	TN	TP	FP	TP	TN	FN	TN	FN	TN
Our Tool	TP	TN	TP	TN	TP	TN	TP	TN	TP	TN

* P: has such a vulnerability, N: no such a vulnerability,
TP: True Positive, FP: False Positive, TN: True Negative, FN: False Negative

Table 3. The summary of comparing with the six non-commercial tools.

Tool Name	Trivial Case	Call Order	Parameter Tracking	Variable Alias	Function Alias
Flawfinder	No	No	No	No	No
VCG	No	No	No	No	No
Cppcheck	No	No	No	No	No
Splint	No	No	No	No	No
Sparse	No	No	No	No	No
RATS	Yes	No	Yes	No	No
Our Tool	Yes	Yes	Yes	Yes	Yes

5.3 Comparing with Existing Detection Methods

We also compared our proposal with six detection methods proposed by existing research activities on TOCTOU detection, including those that benefit from parameter tracking [3, 26-29] or symbolic execution [30]. The results highlight the advantages over others. Table 4 lists the approach these methods adopt, how they explore, and the four features our proposal addresses. The column approach indicates that the method belongs to static code analysis or dynamic code analysis. The column exploration explains how the method checks the code: checking all the code from the beginning to the end without taking branches into account, checking only the current execution path, or finding all possible execution paths. The column call order notes whether the method considers the order of function calls or not. The column parameter tracking shows whether the method tracks the parameters in functions or not. The columns variable alias and function alias are the ability to recognize the aliases to variables and functions.

Table 4. The summary of comparing with existing research activities.

Detection Method	Approach	Exploration	Call Order	Parameter Tracking	Variable Alias	Function Alias
Bishop and Dilger, 1996	Staitc	from beginning to end	No	Yes	No	No
Viega et al., 2000	Static	from beginning to end	No	Yes	No	No
Cowan et al., 2001	Dynamic	single execution path	Yes	Yes	Yes	Yes
Wei and Pu, 2005	Dynamic	single execution path	Yes	Yes	Yes	Yes
Pu and Wei, 2006	Dynamic	single execution path	Yes	Yes	Yes	Yes
Lai, 2018	Static	all possible paths	Yes	No	No	Yes
Our Proposal	Static	all possible paths	Yes	Yes	Yes	Yes

The results show the advantages of our proposal over other detection methods. The first two methods (Bishop and Dilger, 1996; Viega *et al.*, 2000) are based on pattern matching, so that they are unaware of language semantics. It implies that they are faster but do not consider call order, variable alias, and function alias. The three methods in the middle (Cowan *et al.*, 2001; Wei and Pu, 2005; Pu and Wei, 2006) adopt the approach of dynamic code analysis. It means they can get runtime information, but they cannot cover all possible execution paths. These five methods mentioned above all take parameter tracking into account, but they do not benefit from symbolic execution. On the other hand, the method proposed by Lai (2018) uses symbolic execution, but it does not apply parameter tracking. Our proposal can be regarded as an improvement on Lai's detection method and further consider parameter tracking. Note that here we only compare them by discussing these features since the implementations of most related work are no longer available so far as we know.

6. CONCLUSIONS

In order to detect the Time-of-Check to Time-of-Use (TOCTOU) security vulnerability in C-like languages, we proposed a novel detection method that combines symbolic execution and parameter tracking based on static code analysis. We concretely imple-

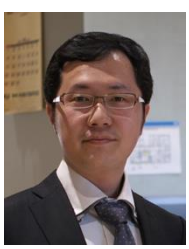
mented a tool for detecting TOCTOU vulnerabilities in C language to show the feasibility and explained the design and implementation. To evaluate our proposal, we first ran our tool with the TOCTOU vulnerability test cases in Juliet Test Suite to verify the essential detection ability. We then evaluated our proposal by comparing it with several non-commercial tools and existing research activities through a set of test cases for highlighting our features. The results show that our proposal has advantages over others in TOCTOU detection, especially in tracking the call order and parameters of functions and recognizing the aliases of variables and functions.

REFERENCES

1. The Software Quality Company, "TIOBE index for July 2021," https://www.tiobe.com/tiobe-index/, 2021.
2. R. C. Seacord, *Secure Coding in C and C++*, Pearson Education, US, 2005.
3. M. Bishop and M. Dilger, "Checking for race conditions in file accesses," *Computing Systems*, Vol. 2, 1996, pp. 131-152.
4. D. Dean and A. J. Hu, "Fixing races for fun and profit: How to use access (2)," in *Proceedings of USENIX Security Symposium*, 2004, pp. 195-206.
5. S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith, "TOCTOU, traps, and trusted computing," in *Proceedings of International Conference on Trusted Computing*, 2008, pp. 14-32.
6. X. Chang, B. Xing, J. Liu, and J. K. Muppala, "LWRM: A lightweight response mechanism for TCG TOCTOU attack," in *Proceedings of IEEE 28th International Performance Computing and Communications Conference*, 2009, pp. 200-207.
7. I. D. O. Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the TOCTOU problem in remote attestation," *arXiv Preprint*, 2020, arXiv:2005.03873.
8. E. Kiciman, B. Livshits, and M. Musuvathi, "CatchAndRetry: Extending exceptions to handle distributed system failures and recovery," in *Proceedings of the 5th Workshop on Programming Languages and Operating Systems*, 2009, pp. 1-5.
9. A. Gosain and G. Sharma, "A survey of dynamic program analysis techniques and tools," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications*, 2015, pp. 113-122.
10. B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, Vol. 35, 2009, pp. 684-702.
11. N. Nethercote, "Dynamic binary analysis and instrumentation," Computer Laboratory, University of Cambridge, 2004.
12. S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of IEEE 24th International Conference on Software Engineer*, 2002, pp. 291-301.
13. T. Ball, "The concept of dynamic analysis," in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundation*, 1999, pp. 216-234.
14. R. D. Venkatasubramanyam and S. GR, "Why is dynamic analysis not used as extensively as static analysis: an industrial study," in *Proceedings of the 1st International*

Workshop on Software Engineering Research and Industrial Practices, 2014, pp. 24-33.
15. M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *Proceedings of Workshop on Dynamic Analysis*, 2003, pp. 24-27.
16. I. Gomes, P. Morgado, T. Gomes, and R. Moreira, "An overview on the static code analysis approach in software development," *Faculdade de Engenharia da Universidade do Porto*, Portugal, 2009.
17. N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, Vol. 25, 2008, pp. 22-29.
18. B. Chess and J. West, *Secure Programming with Static Analysis*, Pearson Education, US, 2007.
19. B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, Vol. 2, 2004, pp. 76-79.
20. T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 1-3.
21. D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Proceedings of IEEE Symposium on Security and Privacy*, 2001, pp. 156-168.
22. W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and System*, Vol. 1, 1992, pp. 323-337.
23. P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977, pp. 238-252.
24. T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java test suite," *IEEE Computer Architecture Letters*, Vol. 45, 2012, pp. 88-90.
25. P. E. Black and P. E. Black, "Juliet 1.3 test suite: changes from 1.2," US Department of Commerce, National Institute of Standards and Technology, 2018.
26. J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," in *Proceedings of IEEE 16th Annual Computer Security Applications Conference*, 2000, pp. 257-267.
27. C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman, "RaceGuard: Kernel protection from temporary file race vulnerabilities," in *Proceedings of USENIX Security Symposium*, 2001, pp. 165-176.
28. J. Wei and C. Pu, "TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, 2005, Vol. 5, p. 12.
29. C. Pu and J. Wei, "A methodical defense against tocttou attacks: The edgi approach," in *Proceedings of International Symposium on Secure Software Engineering*, 2006.
30. T.-C. Lai, "The TOCTOU detection using LLVM static analyzer," Master Thesis, National Chung Cheng University, 2018.
31. G. Tsudik, "Proofs or remote execution and mitigation of TOCTOU attacks," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications*, 2020, pp. 2-3.

32. S. Zeitouni *et al.*, "ATRIUM: Runtime attestation resilient under memory attacks," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, 2017, pp. 384-391.
33. R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, Vol. 51, 2018, pp. 1-39.
34. C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, Vol. 56, 2013, pp. 82-90.
35. E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), " in *Proceedings of IEEE Symposium on Security and Privacy*, 2010, pp. 317-331.
36. L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, 1976, pp. 215-222.
37. J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, Vol. 19, 1976, pp. 385-394.
38. Z. Xu, T. Kremenek, and J. Zhang, "A memory model for static analysis of C programs," in *Proceedings of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, 2010, pp. 535-548.
39. A. Fatima, S. Bibi, and R. Hanif, "Comparative study on static code analysis tools for c/c++," in *Proceedings of IEEE 15th International Bhurban Conference on Applied Sciences and Technology*, 2018, pp. 465-469.



YungYu Zhuang (莊永裕) received the B.S. and M.S. degrees in Mechanical Engineering and Computer Science from National Taiwan University in 2002 and 2004, respectively, and the Ph.D. degree in Information Science and Technology from the University of Tokyo, Japan, in 2014. From 2014 to 2016, he was a Project Assistant Professor with the University of Tokyo. He is currently an Assistant Professor with the Department of Computer Science and Information Engineering, National Central University, Taiwan. He was a Research Assistant with the Central Weather Bureau, Taiwan, from 2004 to 2006, and worked as a Software Engineer in the industry from 2006 to 2011. His research interests include programming language design, software engineering, high-performance computing, machine learning, and programming education.

Yao-Nang Tseng (曾耀儂) completed his master degree in Computer Science and Information Engineering from National Central University. He is interested in writing secure codes and software engineering.