Received: 25 February 2020 | Revised: 13 June 2020 | Accepted: 13 August 2020

DOI: 10.1002/cpe.6004

WILEY

SPECIAL ISSUE PAPER

Static Detection of File Access Control Vulnerabilities on Windows System

Jiadong Lu[1] | Fangming Gu[2] | Yiqi Wang[1] | Jiahui Chen[3] | Zhiniang Peng[4] | Sheng Wen[5]

[1] School of Computer Science and Engineering, South China University of Technology, Guangzhou, China

[2] University of Chinese Academy of Sciences, Beijing, China

[3] School of Computers, Guangdong University of Technology, Guangzhou, China

[4] QiHoo 360, Beijing, China

[5] MIEEE School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Victoria, Australia

Correspondence
Zhiniang Peng, QiHoo 360, Jiuxianqiao Road, Chaoyang District, Beijing 100025, China.
Email: jiushigujiu@gmail.com

Summary

Traditional applications have been developed for decades. Most of the security research around them have focused on the detection of memory corruption vulnerabilities, such as buffer overflow, double fetch, and integer overflow. On the contrary, logic bugs, a kind of flaws caused by unreasonable application logic, attract much less attention. Files are the most common media for programs to persist their data in the system. As the file owners, programs are responsible for protecting their files from malicious users' tampering by leveraging access control mechanisms. However, if a program configures their access control mechanisms in wrong ways and causes evil users to bypass security checks to access files, there exists a file access control vulnerability. As a branch of logic flaws, file access control vulnerabilities are less popular with researchers. Thus, to mitigate the harm of the file access control vulnerabilities on Windows system, our team conducted first-step research on them. We first classified file access control bugs into two types and codified some bug patterns. Then we formalized file access control vulnerabilities to propose a scalable detection method and implemented a lightweight analysis system StaticFAC. After evaluating StaticFAC in real-world Windows software, we discovered 15 0-day bugs.

KEYWORDS

logical vulnerability, static analysis, vulnerability detection

1 | INTRODUCTION

Followed by invading victims' computers, hackers usually run with restricted permissions and are limited to perform a subsequent attack. Hence, most hackers pursue the bugs in traditional desktop software that can help them elevate their privileges. In the old days, these vulnerabilities often resided in operating system kernels. But with the development of operating systems overs these years, kernel defects are becoming increasingly rarer and harder to use. According to a statistical analysis of the Microsoft Security Vulnerability Announcements from January to August 2019[1], we find that hackers concentrated on file-related access control flaws. In that period, Microsoft Security Center patched 557 vulnerabilities on Windows system. One hundred and ten bugs can be leveraged to escalate privileges, and more than two-thirds of them are related to file access control.

Previous security research around traditional desktop software mainly focuses on the triggering and identification of memory corruption bugs. Buffer overflow, integer overflow, and double fetch[2-9] are the typical representatives of this class of bugs. They primarily stem from the out-of-bounds use of variables in computer memory. A classic example is programs that use memory buffers after freeing them, which may cause attackers to control programs after tampering with the values of memory pointers.

One reason for studies in the past focused on memory corruption bugs is the cause and discovery of file access control vulnerabilities are more complicated. Specifically, file access control flaws usually occur in situations where high-privileged processes do not identify callers or handle file paths out of meticulous checks. For instance, suppose there is a privileged process on Windows system that copies files from a file path *A* to another file path *B* at a fixed cycle. And due to a file access control flaw, attackers can refer file path *A* to the file objects they control and file path *B* to critical system executable files. Next, when the privileged process executes its job to copy files again, the system executable file refereed by file path *B* will be replaced with attackers' files, which can help attackers take complete control of computers conveniently. The exploitation of the bug may sound succinct and easy. But actually it requires researchers to gain an in-depth knowledge of the computer file system and permissions mechanism. Besides, this kind of knowledge is not universal and is tied to a specific operating system, which results in a higher research threshold.

To fill the academic gaps in file access control bugs, our team has explored such kind of vulnerabilities. After a study of file access control bugs disclosed by Google Project Zero[10], we observe that in most cases, file access control flaws were caused by Windows services' incorrect use of the system APIs concerning file operations and the permission mechanism. These wrong API usages reflect some program developers lacked security knowledge of the related APIs. According to this kind of behavioral characteristic, we codify some API usage patterns that induce file access control bugs. Based on the features of file access control flaws, we classify them into two categories, that is, security context misuse and check bypass. All of those contents are illustrated in Section 2.

One straightforward approach to detect file access control bugs in Windows binaries at scale is to match their API usage context with our bug patterns summarized before. Nevertheless, this method will encounter two problems. The cause of **the first problem** is that most developers usually wrap system APIs with their functions for the sake of code reuse. Thus, direct invocations of system APIs are hard to appear in one function simultaneously, which results in a sizeable false-negative rate when employing this method to detect bugs. Besides, due to the thousands of APIs on Windows system, performing this bug detection method will lead to an exponential growth in the number of matching rules and reduce our analysis efficiency significantly (**the second problem**).

To address these problems, we adopt a role-oriented analysis method by formalizing file access control bugs. As for an API or a function, it is classified into a corresponding function role based on its feature. In this way, all functions, including those APIs, are mapped to a limited set of function roles. Then, an API and the function wrapping this API have the same function role and can be considered identical, thereby solving **the first problem**. Furthermore, the function analysis is performed for these relatively small number of roles. Hence, the analysis efficiency is improved, and **the second problem** is settled at the same time. In Section 3, we will introduce more critical details about the formalization of file access control bugs to assist in the role-oriented analysis.

In order to detect file access control vulnerabilities practically after formalizing them, we implement a static lightweight system StaticFAC. StaticFAC has the ability to handle multiple programs in one run so that bugs hidden between programs can be detected as well. The workflow of StaticFAC is divided into three steps. In the first step, StaticFAC adopts static backward slicing techniques[11-16] in the evaluated binaries to parse key function parameters, which help to resolve the dependency relationship between the binaries[17-21]. After that, StaticFAC works out a single-binary analysis order for the binaries.

The second step is the core of StaticFAC and is performed for every single binary. Shortly, it first builds a function call graph[22-25] for the program being analyzed and then analyzes each function in the graph. In this step, we have integrated some technologies. For instance, we code a lightweight symbolic execution[26-28] for resolving indirect calls in binaries to create function call graphs.

After performing the single-binary analysis on every binary, a report about them is generated in the third step to demonstrate potentially vulnerable functions and corresponding function call chains. To better assist reverse analysts in concentrating on core targets to confirm the existence of bugs, we propose a risk factor metric to evaluate the function call chains.

In the end, we applied StaticFAC to 9 real-world programs on Windows system and found some bugs. As of writing, 15 of them are marked with CVEs. Such results show the ability of StaticFAC to expose programs' defects. Besides, we still considered the scalability of StaticFAC during the designing and coding of it. If another file access control vulnerability pattern is discovered, we can formalize the bug pattern to scale StaticFAC to catch more bugs.

In summary, the contributions of our paper are:

1. We classify file access control bugs into two types, that is, security context misuse and check bypass, and summarized some vulnerability patterns for each one.

2. By mapping functions in the analyzed programs to function roles and introducing two role operations, we formalize file access control vulnerabilities to propose a scalable method for detecting them at scale.

3. We present the design and implementation of StaticFAC, a lightweight static analysis system to detect file access control vulnerabilities via function call graphs, control flow graphs, and the formal file access control flaw.

4. We evaluate StaticFAC in nine programs that all are produced by Microsoft Corporation and find 15 0-days.

The rest of this article is organized as follows. We classify file access control vulnerabilities and introduce some patterns of them in Section 2. The formalization of file access control vulnerabilities is described in Section 3. Later, we present the model overview of StaticFAC in Section 4 and introduce the design of each component of StaticFAC in Sections 5 to 7. We evaluate StaticFAC in Section 8 and discuss the shortcomings and future work about it in Section 8. Finally, we survey related work in Section 10 and present a conclusion in Section 11.

2 | FILE ACCESS CONTROL VULNERABILITIES

Computer files are widely used for programs to persist their data in the system. They are shared among users on the computer. Thus, privileged programs should take access control of files by leveraging operating system mechanisms, such as the system identity authentication and file system permission control, to prevent malicious users from tampering files. In general, access control involves three elements, that is, objects, subjects, and allowed operations. The first step in its workflow is to identify the subject to which the user belongs. Next, it checks the validity of objects and grants the user access to the object when desired operations are allowed. Based on this characteristic, we divide file access control bugs into two categories, namely, security context misuse and check bypass. In the rest of this section, we exemplify each bug type along with corresponding bug patterns.

2.1 | Security context misuse

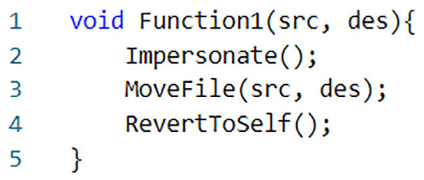
Client/server application scenarios are common occurrences in software engineering. In some cases, clients may request servers to perform file operations. At this time, servers should manipulate files with the permission of clients (that is under their security contexts). Otherwise, if the client is not allowed to access the file in the system, but the server has, security problems may arise.

To alleviate this task, Windows system proposed an impersonation mechanism, which enables servers to simulate the security contexts of clients. In this way, security checks of servers' file operations are performed against clients. Function1 in Figure 1 shows a normal usage of the impersonation mechanism (for the convenience of description, we omit returned value checks and unnecessary function parameters and modify function names).

At the beginning of Function1, the call to Impersonate function converts the server's security context to the client's. At this time, if the client does not have permission to the files specified by *SRC* and *DES* paths, MoveFile function fails with access denial. Later, RevertToSelf function is called to cancel the server's simulation of the client's security context. With the impersonation mechanism, Windows system security can be guaranteed to some extent. However, when developers employ the mechanism without enough understanding of it, file access control flaws may be caused as well. Let us see Function2 shown in Figure 2.

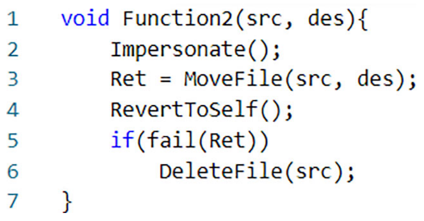
According to the code, when a file path that the client cannot access is passed in, MoveFile function returns fail. And after the service invokes RevertToself function to undo client impersonation, the file specified by *src* variable is deleted under the server's security context. As servers usually run with the highest privilege level in the system, attackers can perform denial of service attacks by abusing Function2 to remove any file. This vulnerability is a type of security context misuse, which means a server manipulates clients' files under wrong security contexts.

To introduce a more realistic bug pattern, we demonstrate a file access control vulnerability that occurs in multiple binaries. We assume there are server *A*, server *B*, and an exported function B_Function of server *B*. And Function1 mentioned above is an exported function of server *A*. Shown in Figure 3, B_Function is leveraged by server *B* to transmit the invocation to Function1 in server *A* for maintaining software compatibility. Unfortunately, this would result in a security context misuse type of file access control bug. This is because the invocation of Impersonate function in Function1 is useless and just renders server *A* to simulate server *B* that also runs with the highest privilege in the system. Thus, malicious clients can cause arbitrary file creation/overwriting damage by simply executing B_Function function.



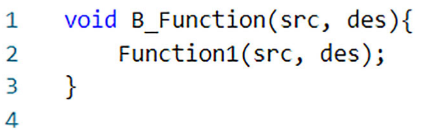
```
void Function1(src, des){
    Impersonate();
    MoveFile(src, des);
    RevertToSelf();
}
```

FIGURE 1 Function1



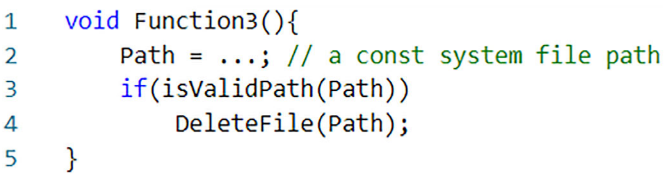
```
void Function2(src, des){
    Impersonate();
    Ret = MoveFile(src, des);
    RevertToSelf();
    if(fail(Ret))
        DeleteFile(src);
}
```

FIGURE 2 Function2



```
void B_Function(src, des){
    Function1(src, des);
}
```

FIGURE 3 B_Function



```
void Function3(){
    Path = ...; // a const system file path
    if(isValidPath(Path))
        DeleteFile(Path);
}
```

FIGURE 4 Function3

2.2 | Check bypass

Under certain circumstances, a server can be triggered by clients to operate system files that have fixed file paths. At this time, the server should perform security checks on the targeted file paths for preventing malicious users from employing file linking technologies to fool the server to manipulate another file. An example is Function3 of Figure 4 (where isValidPath checks whether *Path* is valid and allowed).

However, there exists a contention window between checking permission and operating files. In conjunction with a symbolic link technology, a malicious client can specify an accessible file path first to go through is ValidPath function. Then before DeleteFile function is executed, it changes *Path* to point to another file in the system, thereby achieving arbitrary file deletion.

3 | FORMALIZE FILE ACCESS CONTROL VULNERABILITIES

With file access control vulnerability patterns introduced in the prior section, we have a basic understanding of them and believe it is easy to find the bugs in those functions. In fact, it is just because the calls to these APIs are directly used in a function for the convenience of description. In real massive system software development, developers normally wrap these APIs with their functions. After multiple encapsulations, flaws are hard to be discovered in one function. Besides, due to a large number of operating system APIs, matching API usage context to detect file access control vulnerabilities will generate plentiful matching rules, and that is not feasible to implement efficient bug detection. To solve the problems, formalizing file access control bugs is required.

3.1 | Function roles and operations

Every function has a corresponding role, and we can use its role in turn to describe it. The formal expression of this is, assuming there is a function X, its role is Y, and a mapping F that can map a function to its role, thus, $Y = F(X)$. For example, Impersonate function introduced in Section 2.1 can help a server simulate a client, then we can use a symbol G_1 standing for the **impersonation role** to describe the function, that is, $G_1 = F(Impersonate)$. Similarly, MoveFile function corresponds to the **File Operation role** G_2, and RevertToSelf function belongs to the **Cancel-Impersonation role** G_3. Nevertheless, until now, we cannot deduce the roles of unknown functions from known function roles. In the following, we introduce two role operations to assist in the role-oriented analysis.

Take Function1 mentioned in Figure 1 as a case. As executing Function1 does not lead to any vulnerabilities, it can be described with the safety role G_4. Furthermore, Function1 just has one control flow path that invokes subfunctions Impersonate, MoveFile, and RevertToSelf sequentially. To formally express the role relationship between Function1 and its subfunctions, we introduce a combination operation with a symbol *Comb*(), which can transform multiple ordered function roles into one, thus $G_4 = F(Function1)$, $F(Function1) = Comb(G_1, G_2, G_3)$, and $G_4 = Comb(G_1, G_2, G_3)$.

However, as for functions with multiple control flow paths like Function2 shown in Figure 2, their roles cannot be obtained just with combination operation. Function2 has two control flow paths. One invokes Impersonate, MoveFile, and RevertToSelf, which is identified to Function1 and can be described with the safety role as well. Besides, the other control flow path should be denoted by the unsafety role G_5 for the possibility of leading a file access control loophole. During the detection of file access control vulnerabilities, the role of Function2 should be represented with the unsafety role G_5 rather than safety role G_4 intuitively. To formally describe this process, we define the priorities of function roles as a criterion for comparing them, and propose the simplification operation with a symbol *Simp*() that can convert multiple disordered function roles into one. The formal representation of the role of Function2 is: assume the unsafe role G_5 has a higher priority than the safe role G_4, then $G_5 = Simp(G_4, G_5)$.

In summary, to generate the role of an unknown function, we firstly collect control flow paths as much as possible. Then based on its subfunctions, the roles of these control flow paths can be calculated with the combination operation. In the end, followed by applying the simplification operation to control flow path roles, the role of the unknown function can be acquired. The formal representation of the above process is, assuming the unknown function is X, its role is Y, it has control flow paths $P_1, P_2, \dots, P_n$. Moreover, P_i invoke subfunctions $X_{1i}, X_{2i}, \dots, X_{mi}$ sequentially. Notably, we call the last matrix role matrix (1c).

$$Y = F(X) \tag{1a}$$

$$= Simp(F(P_1),\ \ F(P_2)\ \ \dots\ \ F(P_n)) \tag{1b}$$

$$= \begin{array}{c} \\ Simp \\ Simp \\ \vdots \\ Simp \end{array} \begin{pmatrix} Comb & Comb & \dots & Comb \\ F(X_{11}), & F(X_{12}) & \dots & F(X_{1n}) \\ F(X_{21}), & F(X_{22}) & \dots & F(X_{2n}) \\ \vdots & \vdots & \ddots & \vdots \\ F(X_{m1}), & F(X_{m2}) & \dots & F(X_{mn}) \end{pmatrix} \tag{1c}$$

3.2 | Combine with file access control vulnerability patterns

For realistic utilization of the formal file access control vulnerability to detect bugs, we need to extract related function roles from bug patterns, set up corresponding role operations, and predefine the roles of common APIs in shared libraries. Here, we take the file access control vulnerability pattern in Figure 2 as an example. There are five kinds of function roles, namely, impersonation, file operation, cancel-impersonation, safety, and danger. And we also predefined some imported functions associating with these function roles. All of them are listed in Table 1.

Afterward, we build corresponding role operations. Equation (2) shows the priority relationships between function roles and is used to perform the simplify operation. And five rules are designed for the combination operations. Equation (3) filters out safety role functions. Equation (4) is used to deduplicate two consecutive file operation roles. Equation (5) means that using functions between an impersonation role function and a cancel-impersonation role function is safe. In the end, we employ Equation (6) to find danger role functions since using functions of file operation roles without impersonation may lead to file access control flaws.

With the understanding of bug patterns, function roles, and role operations, we can see that a file access control vulnerability essentially corresponds to a kind of function role and can be generated by other function roles with combination operations. Formalizing file access control flaws transfers the traditional function analysis process into function role analysis, which stands for a higher analysis gain level. Furthermore, a significant feature of the formal file access control vulnerability is scalability. The implementation of it requires a bug pattern to predefine appropriate associated roles and role operations, but it is not limited to the specific patterns presented above. It can scale to another bug pattern if related function roles and operations can be abstracted out from the pattern.

$$Y_5 > Y_2 > Y_1 > Y_3 > Y_4 \tag{2}$$

$$0 =< Y_4 > \tag{3}$$

$$Y_2 =< Y_2, Y_2 > \tag{4}$$

$$Y_4 =< Y_1, Y_2, Y_3 >=< Y_1, Y_5, Y_3 >=< Y_1, Y_3 > \tag{5}$$

$$Y_5 =< Y_2, Y_4 >=< Y_4, Y_2 > \tag{6}$$

TABLE 1 An example of combining a bug pattern and the formal file access control vulnerability

Roles	Symbols	Imported functions
Impersonation	Y_1	Impersonate
File Operation	Y_2	MoveFile, Deletefile
Cancel-Impersonation	Y_3	RevertToSelf
Safety	Y_4	
Danger	Y_5	

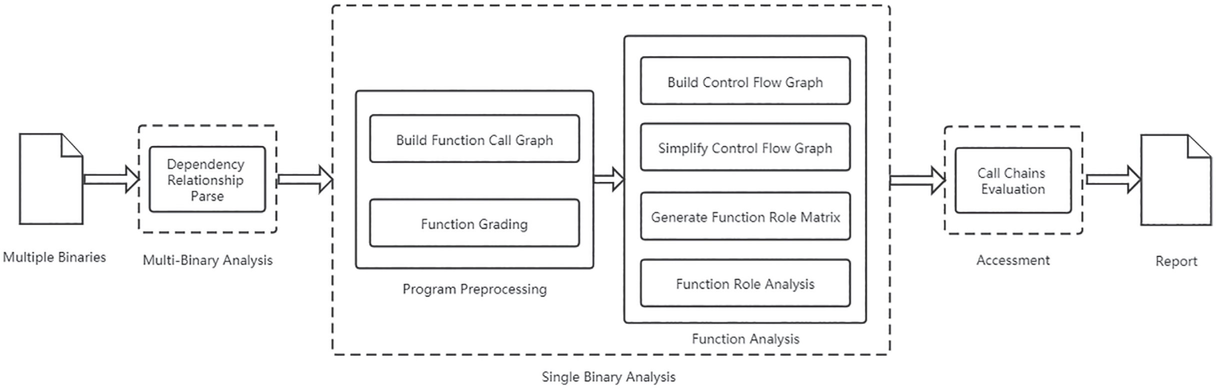


FIGURE 5 An overview of StaticFAC

4 | MODEL OVERVIEW

The formal file access control vulnerability indicates an automatic way to analyze programs and detect file access control bugs. To utilize it practically, we implemented a static and lightweight analysis system StaticFAC. Its architecture is shown in Figure 5 and is divided into three steps, namely, multi-binary analysis, single binary analysis, and assessment. Besides, as the core of StaticFAC is the formalization of file access control bugs, we should go through a configuration part to ensure its normal running, which is mainly introduced in Section 3.2, including function roles, operations, and predefined imported functions.

In short, the purpose of the step of multi-binary analysis is to work out a single-binary analysis order by handling the dependency relationship between analyzed binaries. The single binary analysis step primarily aims to determine the roles of exported functions in the targeted programs. Then StaticFAC assesses its prior work by an empirical criterion in the last step and presents the file access control bugs it detected.

Comparing with other steps, single binary analysis is relatively more complicated and more important. It adopts a greedy strategy to analyze functions in the target program. Starting from underlying imported functions in the binary, it calculates the roles of unknown functions based on known functions. And this process will be iterated layer by layer until StaticFAC handles every exported function of the program. The overall operating process of this step is divided into two parts. The first one is program preprocessing, and the other is function analysis.

The first step of program preprocessing is to create a function call graph[22-25] of the program for gathering as many functions as possible. Functions are regarded as nodes of the function call graph, and the calling relationships between them are denoted as directed edges of the nodes. Afterward, StaticFAC grades functions in the call graph and divides them into multiple sets, so that functions in the same set can be processed in parallel and improves the efficiency of StaticFAC.

After program preprocessing, every function will be analyzed. As for a function, its control flow graph is generated at first. Then some unrelated edges and notes are removed to simplify it. Later, we traverse the control flow graph by a depth-first search, and the results of this process constitute the role matrix (1c). Finally, the function's role can be obtained by calculating the role matrix with role operations.

5 | MULTIBINARY ANALYSIS

In software engineering, developers have made efforts to improve the efficiency of software development. A critical approach is to increase the reuse of codes. Hence, a program or a binary frequently utilizes the exported functions of another binary to take advantage of its capability. Especially, exported functions range a broad spectrum on Windows system. They include not only the functions of traditional dynamic-link libraries, but also unique invocation ways on Windows system, such as RPC, ALPC, and COM. This development model brings a dependency relationship between binaries. Then when a program is specified to execute StaticFAC, StaticFAC must distinguish the program's dependent binaries, thereby covering more targets to analyze.

The multi-binary analysis step begins with employing PE file parsing technology to obtain the program's imported libraries that are recorded in the program file header. Then, StaticFAC looks for the locations that invoke vital functions, such as LoadLibary, GetProcAddress, and NdrClientCall. Moreover, it employs a backward slicing technology[11-15] for parsing these function parameters statically to acquire dependent binaries and corresponding exported functions. Finally, StaticFAC will repeat this process three times for newly obtained binaries to determine modest targets.

Despite the fact that this step does not involve many techniques, some annoying issues were encountered in applying it to binary programs rather than source codes. The most troublesome one is to learn the underlying implementation of function calling technologies and master their data structures and algorithms in assembly. Besides, since analyzing programs is expensive, we leveraged a database to record the information of analyzed programs, which can be provided with the analysis of other programs that import them.

In the end, followed by identifying dependency relationships between targeted binaries, StaticFAC will constantly choose the program with the least dependence relationship to start the single-program analysis part.

6 | PROGRAM PREPROCESSING

Program preprocessing is the first step of analyzing a single binary, including the generation of a function call graph[22-25,29] and function grading. With this step, the sets that consist of unknown functions are obtained.

6.1 | Generate function call graph

A function call graph usually is used for software analysis and reverse engineering. Each node of it stands for a function, and each edge stands for calling relations between them. If there are two nodes F_1, F_2, and an edge is pointed by F_1 to F_2, meaning that function F_1 invokes functions F_2. There are usually two ways to generate the graph, one is a bottom-up method, and the other is top-down. Because in the configuration part, some underlying imported functions have been predefined, then the graph can be built by using the bottom-up method to gather functions that invoke imported functions directly or indirectly. Besides, if there is another internal function that does not relate to those predefined imported functions in the program, it will be ignored. Thus the function call graph may be incomplete but suitable for our file access control flaws detection.

Specially, we use a pair <*ParentFunc*, *ChildFunc*> to represent an edge, where *ParentFunc* calls *ChildFunc*. Moreover, a function call graph in code is a collection of these edges. The Algorithm 1 can generate a function call graph Set_{CG} with the predefined imported library functions as input.

Algorithm 1. Build function call graphs

```
1: procedure BuildCFG($Set_{predefine}$)                ▷ The pre-defined imported functions
2:     $Set_{CG} \leftarrow \emptyset$                                   ▷The output function call graph
3:     $Set_{import} \leftarrow GetImportFunctions()$
4:     $Set_f \leftarrow \emptyset$
5:     $Set_{temp} \leftarrow \emptyset$
6:     for $Func \in Set_{import}$ do
7:         if $Func \in Set_{predefine}$ then
8:             $Set_f.add(F)$
9:         end if
10:    end for
11:    while $Sizeof(Set_f) > 0$ do
12:        $Set_{temp} \leftarrow Set_f$
13:        $Set_f \leftarrow \emptyset$
14:        for $ChildFunc \in Set_{temp}$ do
15:            $Parents \leftarrow GetFunctionParents(ChildFunc)$
16:            for $ParentFunc \in Parents$ do
17:                $Set_f.add(ParentFunc)$
18:                $Set_{CG}.add(<ParentFunc, ChildFunc>)$
19:            end for
20:        end for
21:    end while
22:    return $Set_{CG}$
23: end procedure
```

In the beginning, all the imported functions of the analyzed program are obtained. They are judged one by one (line 4). If it belongs to the predefined functions, it is added into variable Set_f, which represents a set of functions to be used to collect parent functions. Afterward, in each outer loop (line 9), we assign Set_f to another temporary variable Set_{temp} and create a new set for Set_f. In each inner loop (line 12), functions in Set_{temp} are traversed, and their parent functions are fetched. Later, these parent functions are appended into *Set_f* for the next loop in line 15. Line 16 records the pair <*ParentFunc*, *ChildFunc*> in *Set_{funcs}*.

In this way, for each outer loop, the parent functions of the original functions set *Set_f* can be obtained to run as a new *Set_f* collection for a next iteration. When all the functions in *Set_{temp}* have been traversed and no parent function is derived, the function call graph is constructed successfully.

6.1.1 | Recursive functions

In software engineering, developers sometimes need to leverage recursive programming skills to code a function that invokes itself. Unfortunately, our algorithm *BuildCFG* will fall into infinite loops when handling recursive functions.

To solve this problem, in the inner loop (below line 16), before adding parent functions into set *Set_f*, we add statements to determine whether <*ChildFunc*, *ParentFunc*>, which represents an opposite calling relationship from *ChildFunc* to *ParentFunc*, is in set *Set_{CG}*. If so, *ChildFunc* is a recursive function, and we should continue to the next cycle (line 16). For example, assume that there are functions *A* and *B*. *A* invokes *B*, and *B* also calls *A*. At the first, function *A* is in *Set_f* and *B* is not. After the outer loop is executed, set *Set_{funcs}* contains a pair <*B*, *A*>. At this time, function *B* is in *Set_f*, but *A* is not. And When the outer loop is being executed again, the parent function *A* of function *B* is obtained. But as <*B*, *A*> is already in *Set_{funcs}*, the loop should be ended in advance.

6.1.2 | Indirect calls

Since most of the detected programs are written in C++ and the dynamic binding technology and virtual function technology are widely applied in C++, indirect calls inevitably take place in the call graph construction as a particular part. They are the core part of implementing program polymorphism in runtime and can significantly improve software scalability. But at the same time, these advantages make some indirect calls can only be resolved during dynamic execution. In 1992, it was proved that building an exact call graph in static analysis is an undecidable problem[30]. Therefore, to resolve indirect calls as much as possible while maintaining the efficiency of StaticFAC, we solved this problem in a conservative manner by employing a lightweight symbolic execution[26-28,31].

Specifically, our symbolic execution is implemented based on the abstract syntax trees generated from assembly codes by IDA Hexrays[32]. The abstract syntax tree traverses the original variables that are distributed on program stacks and heaps into unified ones, which are similar to variables in high-level languages. This process dramatically improves our programming efficiency and renders it convenient to apply our memory model in the unified variables, explore the relationship between the variables, and parse out the targets of indirect calls.

Besides, as constructing abstract syntax trees and applying symbolic execution to functions are expensive, StaticFAC will identify appropriate functions as targets for analysis. C++ virtual function invocations are common occurrences in indirect calls. Thus, we go deep into the parsing of virtual function calls in the following.

In the beginning, our solution leverages a heuristic method to identify the head addresses of virtual function tables and rebuild the corresponding virtual function tables. To illustrate this method, here we give an example, assuming there has two program memory addresses *A* and *B* and a function *C*. *B* is the head address of the function that is not invoked by any other functions directly. And *B* is stored in address *A* and is referred by function *C*. At this time, we think that function *C* is a class constructor and address *A* is the head of a virtual function table that can be reconstructed via a download scan of address *A*. By employing this method, we can obtain class constructors in the program.

In the next step, we collect functions involving indirect calls by handling all assembly instructions in the program. If the opcode of an assembly instruction corresponds to "call," and the operand is a register or the address of *dispatch_call* function, this instruction is deemed to be an indirect call, and the function in which the instruction resides is recorded.

Afterward, StaticFAC gathers the functions obtained in the first two steps, finds out their common parent functions, and applies our symbol execution in these functions to parse indirect calls.

6.2 | Function grading

According to our formal access control bug, analyzing a function requires the function roles of its subfunctions. Thus, for preventing the situation that a function being analyzed has unhandled subfunctions, we grade functions with the following definition: If a function does not have any subfunctions, its grade is 0. Otherwise, its grade is the highest grade of its subfunctions plus one.

With function grading, we can determine the order of function analysis. For example, suppose we have three functions, *A*, *B*, and *C*. Function *A* uses function *B* and *C*, and function *B* also calls function *C*. Then, we can know that the grade of function *C* is 0, function *B* is 1, and function *A* is 2. And the order of analysis should be *C*, *B*, *A*.

In StaticFAC, we use Algorithm 2 to grade functions. Its input Set_{CG} is the function call graph generated in the previous step, which consists of <*ParentFunc*, *ChildFunc*> pairs. Its output Set_{funcs} is an array, and each element of it is a collection of functions with the same grade. Set_{record} is a secondary dictionary variable used to record all function's grades.

Algorithm 2. Function grading

```
1: procedure FunctionGrading($Set_{CG}$)          ▷ the set of the function call graph generated before
2:     $Set_{funcs} \leftarrow \emptyset$                                   ▷ the output set of grades of functions
3:     $Set_{record} \leftarrow \emptyset$
4:     for $<ParentFunc, ChildFunc> \in Set_{CG}$ do
5:         if $ChildFunc \notin Set_{record}$ then
6:             $Set_{record}[ChildFunc] \leftarrow 0$
7:             $Set_{funcs}[0].add(ChildFunc)$
8:         end if
9:         $ChildGrade \leftarrow Set_{record}[ChildFunc]$
10:        if $ParentFunc \notin Set_{record}$ then
11:            $Set_{record}[ParentFunc] \leftarrow ChildGrade + 1$
12:            $Set_{funcs}[ChildGrade + 1].add(ParentFunc)$
13:        else
14:            $ParentGrade \leftarrow Set_{record}[ParentFunc]$
15:            if $ParentGrade \leq ChildGrade$ then
16:                $Set_{funcs}[ParentGrade].remove(ParentFunc)$
17:                $Set_{funcs}[ChildGrade + 1].add(ParentFunc)$
18:                $Set_{record}[ParentFunc] \leftarrow ChildGrade + 1$
19:            end if
20:        end if
21:    end for
22:    return $Set_{funcs}$
23: end procedure
```

In line 4, for each pair <*ParentFunc*, *ChildFunc*>, if the called function *ChildFunc* is not contained in Set_{record}, *ChildFunc* function has not been analyzed before, the grade of it should be 0, and it has to be placed in the set of grade 0, that is, $Set_{funcs}[0]$. In line 9, we get the grade of *ChildFunc*. Afterward, function *ParentFunc* is analyzed as well. If *ParentFunc* is not in Set_{record}, its grade is directly set to ChildGrade plus one. Otherwise, we have to check whether its current grade is lower than or equal to ChildGrade. If so, remove it from the function grade it was originally in and set its grade to ChildGrade plus one.

By the function grading step, functions are divided into sets in grades, and they will be analyzed one by one to get their roles from the lowest grade 0. In addition, since there is no call relationship between the functions in the same grade, they can be analyzed in parallel that improves the overall efficiency of StaticFAC.

7 | FUNCTION ANALYSIS

In this part, StaticFAC generates a control flow graph (CFG) for every function and simplifies it for the following analysis efficiency. A CFG is a directed graph in which each node represents a basic code block and does not contain jump instructions. And each edge of it represents the control flow relationships between blocks[33-37]. A CFG generally has one starting block and multiple ending blocks. After constructing it, we traverse it from its starting block to its ending blocks, and every result of this process represents a control flow path. Followed by obtaining all control flow paths, we create a subfunction call matrix in which each column represents the subfunctions that a control flow path invokes in sequence. Afterward, through a conversion from function names to function roles, the subfunction call matrix is converted into the role matrix mentioned in Equation (1c). In the end, we calculate the role of the analyzed function by using Equation (1).

7.1 | Build control flow graph

In the traditional CFG, each basic block is filled with codes. But during detecting file access control vulnerabilities, we are concerned with the subfunctions called on each control flow path. Thus, we modify the CFG and save the names of subfunctions in the basic block instead of program instructions. By this step, we can speed up the traversal of a CFG. In Appendix B, we demonstrate the recursive descent algorithm that we use to generate a function's CFG.

7.2 | Simplify control flow graph

With a CFG of a function, we can analyze the function based on the CFG. But before that, we need to consider the efficiency of traversing the CFG. If we have a function that contains more than 20 conditional branches (it commonly takes place in a function that verifies the return values of its invoked subfunctions accurately), there will be more than 2^{20} control flow paths in the function's CFG and the analysis of the CFG will take up a lot of time. As a result, we have to simplify the CFG and remove its useless branches. There are mainly two ways to optimize the graph:

1. If a subfunction does not belong to the previous function call graph, it has little to do with file access control bugs, and its function name will be removed.

2. If there are not any other subfunctions recorded in a basic block, the basic block will be merged with its previous block.

Employing two methods ultimately reduces the CFG's complexity, which is used to verify reachability, relationship with other nodes, edges, and loops effectively[38-41].

7.3 | Generate role matrix

Simplified control flow graphs reduce the number of basic blocks and improve traversal efficiency, but it is not enough. If there is a loop in the function, such as using "while" statements, the traversal will fall into it and constantly generate control flow paths, which eventually leads to the early termination of analysis.

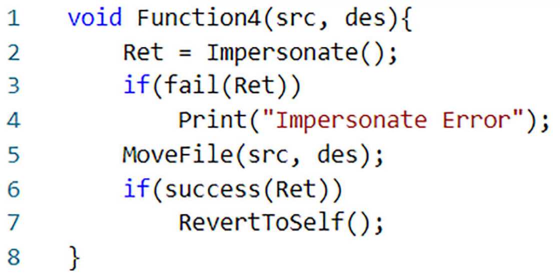
However, when functions execute a loop, the called subfunctions are often the same. Therefore, the roles of the subfunctions in a loop can be considered invariant. Besides, the effects of the same consecutive function roles are equivalent to the impact of one, so in our solution, we unroll the loop once during the traversal of it.

A depth-first search strategy is employed to traverse the control flow graph from the starting block. And the subfunctions in each block encountered during the traversal are recorded in an array with an orderly manner. When the search reaches the ending block, the recorded array is added to the subfunction call matrix. Therefore, when the entire control flow graph is traversed, a subfunction call matrix is formed, which can be converted into the corresponding role matrix by mapping subfunctions' names to their roles.

7.4 | Function role analysis

Having the role matrix, we can generate the role of the analyzed function with role operations. However, as we build control flow paths by traversing the simplified control flow graph in a depth-first search manner, some of the control flow paths may be wrong or unreachable. This problem also appears in other static tools and usually is solved by calculating analytical expressions[2], but it requires a lot of time. Here, we employ a simple but effective heuristic to deal with this problem. Consider Function4 in Figure 6.

Function4 is similar to Function1 shown in Figure 1, except that it checks the return value of Impersonate function. When Impersonate function runs successfully, Function4 executes RevertToSelf function to cancel the client impersonation at the end. Otherwise, Function4 outputs a prompt message. Let us take an insight of the simplified control flow graph of Function4 in Figure 7 that is generated by our prior methods. This graph contains two control flow paths. The first one is Impersonate -> MoveFile -> RevertToSelf -> End, and the other one is Impersonate -> MoveFile -> End. Nevertheless, a client who triggers a server to execute this code can only get the result of the first control flow path. The second path is unreachable and has to be discarded.



```
void Function4(src, des){
    Ret = Impersonate();
    if(fail(Ret))
        Print("Impersonate Error");
    MoveFile(src, des);
    if(success(Ret))
        RevertToSelf();
}
```

FIGURE 6 Function4

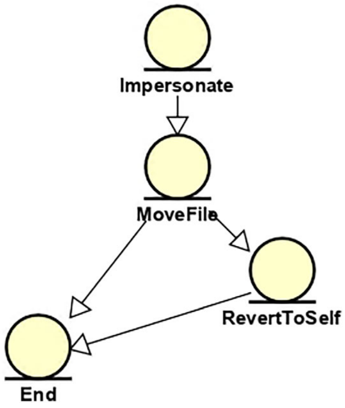


FIGURE 7 The simplified control flow graph of Function4

In fact, the formal file access control vulnerability analyzes functions based on the combinations of function roles, which can be considered as programming modes intuitively. Therefore, in Section 3.2, we should define the combination operations comprehensively, so that we can simplify each column of the role matrix by continually matching them with the combination operations. However, when the roles of a column cannot be calculated, it means that the combination of those roles is not in our analysis patterns and should be discarded, just like the second control flow path.

Followed by calculating all the columns of the role matrix, we will acquire a row of roles that can be parsed to the role of the analyzed function by the simplify operation.

7.5 | Pre-analysis to further optimize efficiency

The analysis of a function is expensive and requires going through the above steps 6. Here we propose a heuristic method to pre-analyze functions and improve the efficiency of StaticFAC. If the combination operation of a function's subfunctions has only one kind of result, the result can be deemed as the role of the function. For example, if a function has subfunctions of file operation roles, its role can only be file operation. Besides, consider a complicated situation that the subfunctions of a function include impersonation roles and cancel-impersonation roles. As functions can only invoke impersonation role subfunctions first and cancel-impersonation role subfunctions in subsequent, the combination operation of the subfunctions gets the result of a safe role.

8 | EVALUATION

8.1 | Risk factor metric

When all functions in the call graph of the analyzed program are handled, we will perform a new analysis of the functions with the highest function grade in the program. This is because most of these functions are exported by the targeted programs and can be directly invoked by analysts to trigger the bugs. If the roles of these functions belong to *Danger*, a report about them is presented. It includes the function call chains, which consist of the functions that are *Danger* role and have invocation relationships with those exported functions.

Unfortunately, even in the absence of considering multi-program analysis, a function call chain generated by a single binary still contains seven functions on average. As a result, this will confuse analysts to target core goals to exploit bugs. To alleviate this problem, we proposed a risk factor metric for evaluating function call chains. The risk factor of a function call chain is calculated by the following rules:

1. The initial value of the risk factor is the lowest function grade among the functions that constitute the function call chain.

2. Except for the lowest grade function, each function on the call chain increases its risk factor by one point. If a function is an exported function of other programs, it will add two extra points.

For instance, there is a short and simple function call chain. Function *A* invokes function *B*, function *B* invokes function *C*, and function *C* invokes function *D*. The function grade of *D* is 2, and *C* is an exported function of another program. Thus, the risk factor of the function call chain is 7. The primary role of the risk factor metric is to assist analysts in focusing on core targets, and the lower risk factor of a function call chain means the easier verification of it.

TABLE 2 Test results

Product	Detected Functions						Exported Functions						Chain	Bug	Function
	All	IMP	FOP	Un-IMP	S	D	All	IMP	FOP	Un-IMP	S	D	Num	Score	Num
SEB	96	2	31	4	47	12	4	0	2	0	1	1	312	12	633
Appx	256	7	76	12	114	47	10	0	3	0	5	2	1231	14	14064
WUPN	9	0	92	0	0	0	3	0	3	0	0	0	22	11	1749
WER	52	1	29	1	21	0	1	0	1	0	0	0	13	7	2716
WIA	133	2	37	17	77	0	3	0	1	0	1	1	38	8	1530
LXSS	76	3	31	1	26	15	4	0	2	0	2	0	214	8	4126
WMS	14	1	7	1	5	0	1	0	1	0	0	0	6	10	5482
MOL	45	2	8	3	30	2	3	0	1	0	2	0	18	12	964
DSS	60	7	36	3	14	0	2	0	2	0	0	0	11	9	411

Abbreviations: (1) Products Names: SEB, System Events Broker; Appx, Windows AppX Deployment Server; WUPN, Windows Update Preparation Notification; WER, Windows Error Reporting Manager; WIA, Windows Image Acquisition; LXSS, Windows Subsystem for Linux; WMS, Windows Media Service; MOL, Microsoft Office OLicenseHeartbeat; DSS, Windows Data Sharing Service; (2) Function roles: Imp, Impersonation role; Fop, File operation role; Un-IMP, Cancel-impersontion role; S, Safety role; D, Danger role.

8.2 Finding

To evaluate the performance of StatcFAC, we applied it to nine latest applications. They all are produced by Microsoft to maintain the normal running of Windows. To prove the applicability of StaticFAC for Windows system software, the programs we picked have diverse features. For example, Windows AppX Deployment Server assists in the deployment of Store applications on local computers, and Windows Error Reporting Manager is responsible for transmitting relevant important information to developers for program optimization when major errors occur.

Before testing StaticFAC, we configured it with the same function roles and operations as described in Section 3.2. And we adapted predefined imported functions to actual vulnerability detection, including six impersonation role functions, four cancel-impersonation role functions, and 21 file operation role functions, which are all shown in Appendix A. These functions were chosen with careful consideration of the efficiency and effectiveness of StaticFAC. And this kind configuration of StaticFAC makes it possible to detect all the bug patterns introduced in Section 2. Given that an exported function is considered a dangerous role, there may be a bug satisfying the vulnerability pattern shown in Figure 2 resides in the function. If another exported function is a file operation role one, the potential bug in it belongs to the flaw pattern in Figure 4.

All the tested programs were compiled as 64-bit Windows PE executables and were publicly released on Windows version 10.0.17763. We evaluated them on a laptop with a 1.70 GHz quad-core Intel Core i5 processor and 8 GB of RAM. The test results are shown in Table 2. In our evaluation, we mainly focus on the total number of detected functions in the programs, the roles of these functions, as well as the same information about exported functions. With those test results and the roles of program functions, reverse analysts can have a general understanding of the distribution of program flaws, and that helps them verify bugs faster. Besides, by taking an insight of the columns Function Num and All in Detected Functions, we can find that they have a large quantity gap, and that means the function call graphs composed by our detected functions are mini. One chief reason for this is the usage of static linked libraries and template libraries in the programs, such as Standard Template Library and Windows Implementation Libraries[42,43], which cause programs to include substantial amounts of library functions. And as most of these functions are not related to the computer file system and access control mechanism, they are filtered by StaticFAC for performance optimization.

After verifying the function call chains, we found 15 0-day bugs, all of which are labeled with CVEs. Shown in Table 3, nine bugs are identified as the type of the check bypass, and the rest of them belong to the security context misuse. Besides, we leveraged the CVSS 3.1 Severity and Metrics[44] to calculate the scores of these bugs. Although the average bug score is 7.3 and some bug scores are not high due to the local utilizing condition, the least harmful one can also cause refusals to work on Windows computers.

TABLE 3 Detected bugs

CVE Identifier	Type	Description	Score
CVE-2019-1253	check bypass	Windows Elevation of Privilege Vulnerability	7.8
CVE-2019-1292	check bypass	Windows Elevation of Privilege Vulnerability	4.9
CVE-2019-1317	security context misuse	Microsoft Windows Denial of Service Vulnerability	7.3
CVE-2019-1340	security context misuse	Microsoft Windows Elevation of Privilege Vulnerability	7.8
CVE-2019-1342	check bypass	Windows Error Reporting Elevation of Privilege Vulnerability	7.8
CVE-2019-1374	security context misuse	Windows Error Reporting Information Disclosure Vulnerability	5.5
CVE-2020-0635	check bypass	Windows Elevation of Privilege Vulnerability	7.8
CVE-2020-0636	security context misuse	Windows Subsystem for Linux Elevation of Privilege Vulnerability	7.8
CVE-2020-0638	check bypass	Update Notification Manager Elevation of Privilege Vulnerability	7.8
CVE-2020-0641	check bypass	Microsoft Windows Elevation of Privilege Vulnerability	7.8
CVE-2020-0697	check bypass	Microsoft Office Tampering Vulnerability	7.8
CVE-2020-0730	security context misuse	Windows User Profile Service Elevation of Privilege Vulnerability	7.1
CVE-2020-0747	check bypass	Windows Data Sharing Service Elevation of Privilege Vulnerability	7.8
CVE-2020-0753	check bypass	Windows Error Reporting Elevation of Privilege Vulnerability	7.8
CVE-2020-0754	security context misuse	Windows Error Reporting Elevation of Privilege Vulnerability	7.8

9 | DISCUSSION AND LIMITATIONS

9.1 | Scalability

Benefited from the formal file access control bug, StaticFAC is scalable and can be extended for detecting other file access control vulnerabilities. When a new file access control vulnerability pattern is discovered, we can extract related function roles, build corresponding role operations, and predefine some APIs in shared libraries that associate with those function roles. After a re-configuration of them in StaticFAC, we can start an automatic detection for this bug pattern.

9.2 | Efficiency and loss of information

The ability of StaticFAC to analyze a function is achieved by establishing a control flow graph of the function and traversing it in a depth-first search manner. Thus, StaticFAC can complete the analysis of a function efficiently. But it also leads to the loss of information with respect to the parameters of invocation to its subfunctions. However, some information is essential and involves files. We have tried to obtain this part of the data to assist in StaticFAC's analysis. Nevertheless, due to the unusually low symbolic execution efficiency and complicated library functions on Windows system, these parameter values are hard to be resolved in the static state.

9.3 | The usage of COM

COM is the abbreviation of Component Object Model, which is introduced by Microsoft in 1993[45,46] and aims at improving the reuse of programs. A program on Windows system sometimes utilizes COM to develop abilities. However, as a core technology of Microsoft, as long as Microsoft introduces a new software development framework, COM will usually have a corresponding new adapted version. This causes the technical details in COM to be bloated and ambiguous. At present, StaticFAC is unable to solve this engineering problem. In the future, we will design analytical algorithms for each COM technical model and establish a database for storing information about them to address this problem.

9.4 | Bug verifications

StaticFAC presents a report about suspicious bugs of a binary after analyzing it. These bugs should be verified for their existence. Nevertheless, as the running environment of programs is complicated, and bugs may occur between multiple binaries, vulnerability verifications are hard to be achieved automatically, and we need to spend an amount of workforce to deal with this problem.

10 | RELATED WORK

10.1 | Access control vulnerabilities

As a subset of access control vulnerabilities, file access control bugs are little known. One reason is there have been many studies on computer access control[47-49], and most of operating systems also provide protection mechanisms for programs, such as identity authentication and file permission control. However, since some developers lack the understanding of computer files and do not have sufficient consideration of symbolic link technologies, file vulnerabilities pose a new threat to operating systems.

Previous research on access control vulnerabilities mostly targets the Web platform. This is due to the natural statelessness of HTTP protocol web applications should verify the identity of every HTTP requester, which is an error-prone practice for developers. For the past 10 years, OWASP has mentioned access control bugs in its TOP 10 Web Application security risks[50]. To reduce the harm of these bugs, many researchers have put effort into it. For instance, Sun et al[51] observed that web pages could represent a kind of program state, and if the transitions between these program states do not have sufficient security checks, access control bugs will be induced. Based on this concept, they build sitemaps of web applications to identify access control bugs. Furthermore, Deepa et al[52] extract the data flow and control flow of web applications to detect more types of vulnerabilities, which include access control bugs.

10.2 | The generation of call graph

A call graph represents calling relationships between functions in a program. The research on it can even date back to 1970s[53]. At present, call graphs are commonly applied in program understanding and malicious program analysis[22-25]. They also play a pivotal role in StaticFAC. A more precise call graph can provide StaticFAC more functions to analyze, thereby strengthening its ability to discover more bugs. However, due to the common occurrence of indirect calls in binary files, building an exact call graph in static analysis has been proved to be an undecidable problem in 1992[30]. In 2019, Schwartz et al[54] proposed a novel solution, which can recover over 78% of methods to the correct class in authors' evaluation via combining a symbolic analysis with a flexible reasoning system. This method can parse indirect calls to a large extent, but it also brings extremely low efficiency. Thus, we chose to leverage the symbolic execution technique to generate call graphs while achieving a good tradeoff between graph precision and construction efficiency. We implemented our symbolic execution based on the decompiled abstract syntax tree generated by IDA Hexrays[32], which help us reconstruct program variables resided on stacks and heaps. Then when creating a call graph of a binary, StaticFAC traverses the program instructions and models the computer memory of encountered variables at the same time. By this step, StaticFAC can explore the relationship between variables and parse out the targets of indirect calls.

11 | CONCLUSION

File access control vulnerabilities are harmful to the system. Unfortunately, few people do research on them. In this article, we presented a lightweight static analysis system StaticFAC and evaluated it on Windows real-world software to show its ability to detect file access control bugs. As of writing, we have discovered 15 bugs that were assigned with CVEs. Moreover, due to the scalability of the formal file access control vulnerability, StaticFAC is also possible to detect another novel file access control vulnerability at scale.

ORCID

Jiadong Lu https://orcid.org/0000-0002-1892-5971
Fangming Gu https://orcid.org/0000-0002-2531-4642
Jiahui Chen https://orcid.org/0000-0001-7128-9778

REFERENCES

1. MSCR. Microsoft Security Response Center Acknowledgements. Website; 2019. https://portal.msrc.microsoft.com/en-us/security-guidance/acknowledgments.
2. Xu M, Qian C, Lu K, Backes M, Kim T. Precise and scalable detection of double-fetch bugs in OS kernels. Paper presented at: 2018 IEEE Symposium on Security and Privacy. San Francisco, CA; IEEE; 2018:661-678.
3. Kroes T, Koning K, van der Kouwe E, Bos H, Giuffrida C. Delta pointers: buffer overflow checks without the checks. Paper presented at: Proceedings of the Thirteenth EuroSys Conference. New York, NY; 2018:1-14.
4. Lemieux C, Sen K. Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. Paper presented at: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. Montpellier, France; 2018:475-485.
5. Dietz W, Li P, Regehr J, Adve V. Understanding integer overflow in C/C++. *ACM Trans Softw Eng Method (TOSEM)*. 2015;25(1): 1-29.
6. Sun H, Zhang X, Su C, Zeng Q. Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability. Paper presented at: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security. New York, NY; 2015:483-494.
7. Sidiroglou-Douskos S, Lahtinen E, Rittenhouse N, et al. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. Paper presented at: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY; 2015:473-486.
8. Gao F, Wang L, Li X. BovInspector: automatic inspection and repair of buffer overflow vulnerabilities. Paper presented at: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. Singapore; 2016:786-791.
9. Liu WX, Cai J, Wang Y, Chen QC, Tang D. Mix-flow scheduling using deep reinforcement learning for software-defined data-center networks. *Internet Techn Lett*. 2019;2(3):e99. https://doi.org/10.1002/itl2.99.
10. Google . Details of vulnerabilities publicly disclosed by Google Project Zero. Website; 2019. https://bugs.chromium.org/p/project-zero/issues/list.
11. Jaffar J, Murali V, Navas JA, Santosa AE. Path-sensitive backward slicing. Paper presented at: International Static Analysis Symposium. Berlin, Heidelberg: Springer; 2012:231-247.
12. Song J, Tørring JO, Hyun S, Jee E, Bae DH. Slicing executable system-of-systems models for efficient statistical verification. Paper presented at: 2019 IEEE/ACM 7th International Workshop on Software Engineering for Systems-of-Systems (SESoS) and 13th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (WDES). Montreal, QC: IEEE; 2019:18-25.
13. Lisper B, Masud AN, Khanfar H. Static backward demand-driven slicing. Paper presented at: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation. New York, NY; 2015:115-126.
14. Srinivasan V, Reps T. An improved algorithm for slicing machine code. *ACM SIGPLAN Not*. 2016;51(10):378-393.
15. Abdallah M, Alokush B, Alrefaee M, Salah M, Bader R, Awad K. JavaBST: Java backward slicing tool. Paper presented at: 2017 8th International Conference on Information Technology (ICIT); IEEE; 2017:614-618.
16. Wang Y, Meng W, Li W, Liu Z, Liu Y, Xue H. Adaptive machine learning-based alarm reduction via edge computing for distributed intrusion detection systems. *Concurr Comput Pract Exper*. 2019;31(19):e5101. https://doi.org/10.1002/cpe.5101.
17. Ito S. Semantical equivalence of the control flow graph and the program dependence graph. *arXiv preprint arXiv:1803.02976* 2018.
18. Johnson A, Waye L, Moore S, Chong S. Exploring and enforcing security guarantees via program dependence graphs. *ACM SIGPLAN Not*. 2015;50(6):291-302.
19. Ishihara N, Funabiki N, Kao WC. A proposal of statement fill-in-blank problem using program dependence graph in Java programming learning assistant system. *Inform Eng Exp*. 2015;1(3):19-28.
20. Ullah F, Wang J, Jabbar S, Al-Turjman F, Alazab M. Source code authorship attribution using hybrid approach of program dependence graph and deep learning model. *IEEE Access*. 2019;7:141987-141999.
21. Yu X, Liu J, Yang ZJ, Liu X, Yin X, Yi S. Bayesian network based program dependence graph for fault localization. Paper presented at: 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). Ottawa, ON: IEEE; 2016:181-188.
22. Wu P, Wang J, Tian B. Software homology detection with software motifs based on function-call graph. *IEEE Access*. 2018;6: 19007-19017.
23. Malik MM, Pfeffer J, Ferreira G, Kästner C. Visualizing the variational callgraph of the Linux kernel: an approach for reasoning about dependencies [poster]. Paper presented at: Proceedings of the Symposium and Bootcamp on the Science of Security. New York, NY; 2016:93-94.
24. Madsen M, Tip F, Lhoták O. Static analysis of event-driven Node. js JavaScript applications. *ACM SIGPLAN Not*. 2015;50(10): 505-519.
25. Hassen M, Chan PK. Scalable function call graph-based malware classification. Paper presented at: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. New York, NY; 2017:239-248.
26. Baldoni R, Coppa E, D'elia DC, Demetrescu C, Finocchi I. A survey of symbolic execution techniques. *ACM Comput Surv (CSUR)*. 2018; 51(3):1-39.
27. Mechtaev S, Griggio A, Cimatti A, Roychoudhury A. Symbolic execution with existential second-order constraints. Paper presented at: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, NY; 2018:389-399.
28. Wang H, Liu T, Guan X, Shen C, Zheng Q, Yang Z. Dependence guided symbolic execution. *IEEE Trans Softw Eng*. 2016;43(3):252-271.
29. Wang W, Li Y, Wang X, Liu J, Zhang X. Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers. *Futur Gener Comput Syst*. 2018;78:987-994. https://doi.org/10.1016/j.future.2017.01.019.
30. Landi W. Undecidability of static analysis. *ACM Lett Program Lang Syst*. 1992;1(4):323-337.
31. Meng W, Li W, Wang Y, Au MH. Detecting insider attacks in medical cyber–physical networks based on behavioral profiling. *Futur Gener Comput Syst*. 2018;108:1258–1266. https://doi.org/10.1016/j.future.2018.06.007.
32. Hex Rays S. IDA Pro Hex rays Decompiler. https://www.hex-rays.com/products/decompiler/. 2019.
33. Dietrich C, Hoffmann M, Lohmann D. Cross-kernel control-flow-graph analysis for event-driven real-time systems. *ACM SIGPLAN Not*. 2015; 50(5):1-10.

34. Nguyen MH, Le Nguyen D, Nguyen XM, Quan TT. Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning. *Comput Secur*. 2018;76:128-155.
35. Chilenski M, Cybenko G, Dekine I, Kumar P, Raz G. *Control flow graph modifications for improved RF-based processor tracking performance*. Vol 10630. Orlando, Florida, United States: International Society for Optics and Photonics; 2018.
36. Jonischkeit C, Kirsch J. Enhancing control flow graph based binary function identification. Paper presented at: Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium. New York, NY; 2017:1-8.
37. Payer M, Barresi A, Gross TR. Fine-grained control-flow integrity through binary hardening. Paper presented at: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Berlin, Heidelberg: Springer; 2015:144-164.
38. Anand RV, Dinakaran M. Improved scrum method through staging priority and cyclomatic complexity to enhance software process and quality. *Int J Internet Techn Secur Trans*. 2018;8(2):150-166.
39. Liu H, Gong X, Liao L, Li B. Evaluate how cyclomatic complexity changes in the context of software evolution. Paper presented at: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC). Tokyo, Japan: IEEE; 2018:756-761.
40. Mohammad CW, Shahid M, Husain SZ. A graph theory based algorithm for the computation of cyclomatic complexity of software requirements. Paper presented at: 2017 International Conference on Computing, Communication and Automation (ICCCA); IEEE; 2017:881-886.
41. Qiu J, Zhang J, Luo W, et al. A3CM: automatic capability annotation for Android malware. *IEEE Access*. 2019;7:147156-147168. https://doi.org/10.1109/access.2019.2946392.
42. Standard Template Library. [Online]. http://www.cplusplus.com/reference/stl/. 2020.
43. Windows Implementation Libraries. [Online]. https://github.com/microsoft/wil. 2020.
44. Common Vulnerability Scoring System version 3.1. [Online]. https://www.first.org/cvss/specification-document. 2020.
45. Rubtcova M, Pavenkov O. Enterprise distributed component object model's architecture of the information system. Paper presented at: International Conference on Advanced Computer Science and Information Technology (ICACSIT), Pune, India; April 29, 2018.
46. Lucco S, Lafreniere L, Qu Y. Dynamic code generation and memory management for component object model data constructs. U.S. Patent No. 10,248,415. 2019.
47. Tourani R, Misra S, Mick T, Panwar G. Security, privacy, and access control in information-centric networking: a survey. *IEEE Commun Surv Tutor*. 2017;20(1):566-600.
48. Levergood TM, Stewart LC, Morris SJ, Payne AC, Treese GW. Internet server access control and monitoring systems. U.S. Patent No. 9,917,827. 2018.
49. Schell SV, Haggerty DT. Management systems for multiple access control entities, U.S. Patent No. 10,200,853; 2019.
50. OWASP . OWASP Top Ten Security Risk. [Online]. https://owasp.org/www-project-top-ten/ 2020.
51. Sun F, Xu L, Su Z. Static detection of access control vulnerabilities in web applications. Paper presented at: USENIX Security Symposium. San Francisco, CA; 2011.
52. Deepa G, Thilagam PS, Praseed A, Pais AR. DetLogic: a black-box approach for detecting logic vulnerabilities in web applications. *J Netw Comput Appl*. 2018;109:89-109. https://doi.org/10.1016/j.jnca.2018.01.008.
53. Ryder BG. Constructing the call graph of a program. *IEEE Trans Softw Eng*. 1979;3:216-226.
54. Schwartz EJ, Cohen CF, Duggan M, Gennari J, Havrilla JS, Hines C. Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY: Association for Computing Machinery; 2018;426-441. https://doi.org/10.1145/3243734.3243793.

How to cite this article: Lu J, Gu F, Wang Y, Chen J, Peng Z, Wen S. Static Detection of File Access Control Vulnerabilities on Windows System. *Concurrency Computat Pract Exper*. 2020;e6004. https://doi.org/10.1002/cpe.6004

APPENDIX A. EXACT CONFIGURATION OF STATICFAC

See Table A1.

TABLE A1 Roles, symbol, imported functions in StaticFAC

Roles	Symbols	Imported functions
Impersonation	Y_1	RpcImpersonateClient, ImpersonateLoggedOnUser, ImpersonateNamedPipeClient, ImpersonateAnonymousToken, CoImpersonateClient, SeImpersonateClient
File operation	Y_2	CreateFile, SetNamedSecurityInfo, SetSecurityFile, SetFileAttributes, CopyFile, CopyFileEx, ReplaceFile, MoveFileEx, MoveFile, MoveFileWithProgress, WriteFile, WriteFileEx, RemoveDirectory, DeleteFile, SetSecurityInfo, ZwCreateFile, ZwWriteFile, SHCreateDirectory, CreateDirectory
Cancel-Impersonation	Y_3	RevertToSelf, RpcRevertToSelfEx, RpcRevertToSelf, CoRevertToSelf
Safety	Y_4	
Danger	Y_5	

APPENDIX B. THE ALGORITHM OF BUILDING A OPTIMIZED CONTROL FLOW GRAPH FOR A FUNCTION

Algorithm 3. Generate a control flow graph of a function

```
procedure GENERATECFG(Address)                          ▷ The starting address of the analyzed function
    $StartBlock.StartAddress \leftarrow Address$         ▷ The start block
    $Blocks \leftarrow \emptyset$                         ▷ The set of blocks in the control flow graph
    $Blocks.add(StartBlock)$
    $Set_{address} \leftarrow \emptyset$                  ▷ The collections of handled addresses
    $HelperFunc(StartBlock, Set_{address}, Blocks)$      ▷ A recursive helper function
    return $Blocks$
end procedure
```

Algorithm 4. HelperFunc - A recursive function helps to build the control flow graph

```
procedure HELPERFUNC(CurrBlock, $Set_{address}$, Blocks)   ▷ CurrBlock stands for the block that is being analyzed by the recurisve function
    $Address \leftarrow CurrBlock.StartAddress$
  while $Address \notin Set_{address}$ do
      $Set_{address}.add(Address)$
      $Code \leftarrow GetAddressCode(Address)$
      if isJumpCode($Code$) then
          $CurrBlock.EndAddress \leftarrow Address$
          $JumpAddress \leftarrow GetJumpAddress(Code, Address)$
          if $JumpAddress \notin Set_{address}$ then
              $NewBlcok1 \leftarrow \emptyset$
              $NewBlock1.StartAddress \leftarrow JumpAddress$
              $Blocks.add(NewBlock1)$
              $CurrBlock.addNextBlock(NewBlock1)$
              $HelperFunc(NewBlock1, Set_{address}, Blocks)$
      else
          $Block2 \leftarrow Blocks.GetBlockByAddress(JumpAddress)$
          if $Block2.StartAddress\ ! = JumpAddress$ then
              $SpiltBlock \leftarrow Block2.SpiltAt(JumpAddress)$
              $Blocks.add(SpiltBlock)$
              $CurrBlock.addNextBlock(SpiltBlock)$
          else
              $CurrBlock.addNextBlock(Block2)$
          end if
      end if
      if isNotConditionaljump(code) then
          $NextAddress \leftarrow GetNextAddress(Address)$
          $NewBlocks2 \leftarrow \emptyset$
          $NewBlock2.StartAddress \leftarrow NextAddress$
          $CurrBlock.addNextBlock(NewBlock2)$
          $GenerateCFG(NewBlock2, Set_{address}, Blocks)$
      end if
      return
else if $isReturn(code)$ then
      $CurrBlock.EndAddress \leftarrow address$
      return
```

```
else if IsCallCode(Code) then
        $CurrBlock.addFunction(CalledFunctionAddress(Code))$
end if
      $Address \leftarrow GetNextAddress(Address)$
     end while
     $CurrBlock.EndAddress \leftarrow Address$
return
end procedure
```