Research Article

Fine-Grained Control-Flow Integrity Based on Points-to Analysis for CPS

Weizhong Qiang , Shizhen Wang, Hai Jin , and Jiangying Zhong

Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

Correspondence should be addressed to Jiangying Zhong; eileen_zjy@hust.edu.cn

Received 27 April 2018; Revised 4 September 2018; Accepted 4 October 2018; Published 17 October 2018

Academic Editor: Gregorio Martinez Perez

Copyright © 2018 Weizhong Qiang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

A cyber-physical system (CPS) is known as a mix system composed of computational and physical capabilities. The fast development of CPS brings new security and privacy requirements. Code reuse attacks that affect the correct behavior of software by exploiting memory corruption vulnerabilities and reusing existing code may also be threats to CPS. Various defense techniques are proposed in recent years as countermeasures to emerging code reuse attacks. However, they may fail to fulfill the security requirement well because they cannot protect the indirect function calls properly when it comes to dynamic code reuse attacks aiming at forward edges of *control-flow graph* (CFG). In this paper, we propose P-CFI, a fine-grained *control-flow integrity* (CFI) method, to protect CPS against memory-related attacks. We use *points-to* analysis to construct the legitimate target set for every indirect call cite and check whether the target of the indirect call cite is in the legitimate target set at runtime. We implement a prototype of P-CFI on LLVM and evaluate both its functionality and performance. Security analysis proves that P-CFI can mitigate the dynamic code reuse attack based on forward edges of CFG. Performance evaluation shows that P-CFI can protect CPS from dynamic code reuse attacks with trivial time overhead between 0.1% and 3.5% (Copyright © 2018 John Wiley & Sons, Ltd.).

1. Introduction

CPS provides a paradigm for managing and controlling interconnected physical devices. The past several years have witnessed the fast development and wide deployment of CPS. It is not ambiguous that CPS plays an increasingly important role in different field such as transportation, health-care, energy conservation, and military. As the pervasion of CPS, the problems concerning security and privacy of CPS also raise attentions. Memory unsafe programming languages like C and C++ that are widely adopted by CPS allow programmers to manage the memory space of applications with no constraints, which brings many security problems to CPS.

The control-flow hijacking attack is a typical memory-related attack that has been a severe threat to the security of CPS and has drawn significant attention from both academic and industrial communities. SeismoMeter [1] combines approximate control-flow integrity, fast dynamic taint analysis, and API sandboxing schemes to recognize both control-flow hijacking and data-only attacks. Code injection

is a typical control-flow hijacking attack, which diverts the control-flow of vulnerable programs through buffer overflow to the injected malicious shellcode that resides in data pages, such as heap and stack. However, the technique of *Data Execution Prevention* (DEP) [2] can prevent this attack by prohibiting code in data pages from executing. Further, *stack canaries* [3] are used against the buffer overflow, and DieHard [4] is applied to protect memory for unsafe languages, so that attackers cannot divert the control-flow of applications.

The code reuse attack is another form of control-flow hijacking technique that may bypass fore-mentioned countermeasures. It transfers the control-flow to instruction sequences existing in code pages. Return-to-libc [5] and Return-Oriented Programming (ROP) [6] are examples of code reuse attack. Most previous mitigation methods aiming at code injection attacks such as DEP [2] fail to fully protect programs from code reuse attacks. As a result, a number of new defense methods have been proposed.

Countermeasures based on control-flow integrity (CFI), such as original CFI [7], forward-edge CFI [8], CCFIR [9],

MCFI [10], KCOCFI [11], FPGate [12], and Bin-CFI [13], ensure the correct behavior of CPS by constraining their control-flow using pregenerated *control-flow graph* (CFG). Works such as *Code Pointer Integrity* (CPI) [14] defend programs against code reuse attacks by protecting all the sensitive pointers inside a program and storing them in hidden locations of memory. *Address Space Layout Randomization* (ASLR) [15] allows different modules in a program to be distributed in random locations of memory when these modules are loaded, which makes it harder for attackers to find gadgets.

However, the new code reuse attacks based on dynamic analysis aiming at attacking forward-edge control-flow equipped with advanced techniques may also bypass these defense methods. The work *Newton* [16] presents a runtime gadget-discovery framework, which can generate code reuse attacks aiming at indirect function calls with different strategies corresponding to different kinds of code reuse defenses using constraint-driven dynamic analysis. It proves that the most stringent defense to date which is based on function type signature has also been bypassed.

In this paper, we explore the characteristics and concept of this kind of attack profoundly. We also revisit the motivation and concept of the state-of-the-art protection methods against new code reuse attacks aiming at protecting forward control-flow and explain why these existing methods fail to prevent software programs against the dynamic code reuse attack achieved by the work *Newton* [16]. In order to protect programs from the new dynamic code reuse attack, we propose a fine-grained CFI method for forward edges of CFG based on *points-to* analysis [17, 18], named P-CFI. P-CFI instruments the legitimate target set for every indirect call site and checks whether the target of the indirect function call is in the legitimate set at runtime. Legitimate target sets of indirect function calls are generated by *points-to* analysis of call site value and function type information.

It is worth pointing out that we only use the existing *points-to* analysis for analyzing the function pointers and virtual table pointers, and we have not made new contribution in the *points-to* analysis itself. Scalability and precision are two contradictory factors that need to be balanced in *points-to* analysis. Performing *points-to* analysis in large programs with sufficient precision is a challenge. Most *points-to* analysis methods focus on scalability, at the expense of precision.

In summary, we make the following contributions in this paper:

- (i) We analyze state-of-the-art works for preventing software programs against code reuse attacks and explain why they suffer from the dynamic code reuse attack achieved by the work *Newton* [16].
- (ii) We propose a fine-grained CFI-based defense method called P-CFI to mitigate the dynamic code reuse attack. We achieve this by protecting the indirect function call and forcing the program to check the transfer of the indirect function call before calling it.
- (iii) We implement a prototype of P-CFI according to the concept we introduced. The implementation of P-CFI can be integrated into LLVM.

- (iv) We do the performance evaluation of the implemented prototype based on LLVM using SPEC CPU2006 benchmarks and do the security analysis on open source project *nginx* to prove the practicability of P-CFI.

The rest of this paper is organized as follows: we discuss the background knowledge and analyze related works in Section 2. The threat model and assumptions are explained in Section 3. The concept and work flow of our proposed defense method P-CFI are introduced in Section 4. The details of design and implementation are described in Sections 5 and 6. Experimental results of performance and security analysis are presented in Section 7. At last, we conclude our work in Section 8.

2. Background and Related Work

In this section, we briefly introduce the concept of CPS firstly. We then introduce the fundamental concept beneath code reuse attacks and the development of them, as well as the corresponding countermeasures, before we introduce the principle of our mitigation method.

2.1. Cyber-Physical System. CPS can be regarded as the intersection of computation and the physical world. It can not only perceive and detect the world through various sensors but also reflect changes through different interconnected actuators. The form and size of CPS are various, from small sensor node to complex large system [19]. CPS can be applied to different areas, such as medical electronic devices, internet of vehicles, aeronautical facilities, military weapons, industrial control systems, smart grids, and other infinite applications.

For many CPS, although the devices are isolated from other undesired external influences in physical way, the continuous growing complexity may also bring new, unexpected attack surfaces. CPS can be vulnerable to attack even when not directly accessible. Technologies such as wireless communication, internet connectivity, and internet of thing can expose CPS to various security threats. We can see that CPS is vulnerable to potential forms of attacks that are the same as those attacks experienced by different web services, personal computers, and interconnected devices [20], which means that flaws of software existing in ordinary systems will also cause failures in CPS [21].

2.2. Code Reuse Attacks. Since the wide deployment of DEP [2], code reuse attacks as another kind of control-flow hijacking attacks have raised wide attention, such as *Newton* [16], fully call-preceded attack [22], stitching the gadgets attack [23], and COOP [24]. Different from code injection attacks, code reuse attacks divert the regular control-flow of programs to gadgets that exist in code pages.

ROP is a common technique of code reuse attack aiming at hijacking control-flow of target programs, which diverts the program's execution flow to a set of instruction sequences that end with return instruction. Each instruction sequence is called a gadget. Usually, a complete ROP attack requires various gadgets, with each of them performing a simple

operation. Defense techniques, such as modern shadow stack [25], original CFI [7], forward-edge CFI [8], CCFIR [9], MCFI [10], KCoFI [11], FPGate [12], Bin-CFI [13], ASLR [15], and TRaP [26], can increase the difficulty of launching a successful ROP attack and therefore ensure the security of target programs.

Another powerful attack method that reaches this goal is the function reuse attack, which reuses the functions in a program. Function pointers and virtual calls are examples of exploiting targets. The destination addresses of them are dynamically determined during runtime. The function reuse attack can be regarded as a variant of ROP attacks. However, defense techniques aiming at ROP may fail to protect target programs from this kind of attack. Counterfeit Object-Oriented Programming (COOP) [24] is one of the examples which launches a successful attack by reusing virtual function calls in the target program. There are numbers of defense methods that have been proposed since the attack was presented. VTrust [27] guarantees that virtual function call site invokes virtual functions with the same name and argument type list and a compatible class relationship, so that attacks cannot use random functions of the program's *vtables*. TypeArmor [28] provides a binary protection mechanism, which uses *use-def* analysis at *callees* and function parameters as constraints to decrease the target functions of the indirect call site. TRaP [26] proposes a method that randomizes the address of code pointers.

However, another work called Newton [16] proposes a forward-edge based code reuse attack by leveraging dynamic analysis that can bypass current defense methods including ASLR, CFI, and CPI. It makes the attack based merely on indirect call realizable for that it can jump to any functions with any constructed argument list, while evading the detection of ROP attack such as shadow stack.

2.3. Control-Flow Integrity. Since memory unsafe language allows programmers to manage the memory of programs freely, it is the programmers' responsibility to make sure the control-flow in a program is transferred in legitimate way. Attackers can divert the control-flow of the program to any executable address through corrupting memory of vulnerable programs.

CFI is a lightweight method to cope with this situation. It ensures the correct behavior of a program by constraining the control-flow using pregenerated CFG of the target program and makes sure every illegitimate transfer is trapped. It can be classified into backward CFI and forward CFI or source code based CFI and binary based CFI. In this paper, we protect target programs based on forward CFI.

As a matter of fact, nearly all CFI methods consisted of two processes, the preprocessing phase which analyzes the program and constructs a CFG that represents the legitimate transfer pattern for control-flow and the enforcement phase that constrains the control-flow according to the CFG.

2.4. Related Work. Previous studies prove that CPS also suffers from memory-related vulnerabilities [21]. In this section, we discuss the evolvement of both the defense and attack methods of memory-related vulnerabilities.

The development of defense methods always keeps pace with the emerging code reuse attack techniques. As for the concept of these defense methods, some of them focus on constraining the transfer target of CFG into a predefined legitimate set, such as original CFI [7], forward-edge CFI [8], CCFIR [9], MCFI [10], KCoFI [11], FPGate [12], and Bin-CFI [13]. On the other hand, some works make it difficult for attackers to find usable gadgets, such as practical ASLR [15], TRaP [26], and TASR [29]. There are also mitigation methods about protecting the code pointers by putting them in hidden memory location, such as CPI [14]. Our work takes the concept of forward-edge CFI and focuses on protecting indirect function-calls of C programs or virtual table pointer dereferencing of C++ programs.

As the attack techniques are evolved from code injection attack to code reuse attack by attackers, the defense methods are also improved. The first work of control-flow integrity appeared in original CFI [7] with a practical implementation in the year of 2005. It is hard to create complete CFG for programs with only binary. So we cannot adequately protect the program without source file. And also full instrumentation of binary file will bring much time overhead. In order to improve the performance of CFI, many works focus on coarse-grained CFI. For example, Bin-CFI [13] and CCFIR [30] attempt to relax constraints on the legitimate target set for both the backward (e.g., *ret* instructions) and forward (e.g., indirect call instructions) edges of CFG. However, *out of control attack* [31] and *stitching the gadgets attack* [23] can still find useful gadgets to bypass the coarse-grained CFI mentioned above.

In general, source code based fine-grained CFI solutions bring more accuracy and less time overhead. Source level fine-grained CFI, such as forward-edge CFI [8] which focuses on protecting forward edges of CFG and CCFI [30] which focuses on protecting control-flow transfers in the CFG by adding message authentication code (MAC) to control-flow elements, can defend *out of control attack* [31] and *stitching the gadgets attack* [23] in practice. However, these defense methods can still be bypassed by attacks such as Control Jujutsu [32] and Control-Flow Bending [33]. Modern shadow stack techniques, such as shadow stacks [25], are developed to defend programs against ROP attack. However, the work in [32] proves that programs with protection of shadow stack are still vulnerable.

Function reuse attacks, such as COOP [24], are variants of code reuse attacks. They reuse functions in target programs. COOP attack reuses virtual functions in *vtables*. Several new defense techniques aiming at protecting functions are presented immediately as countermeasures. For example, VTrust [27] and TypeArmor [28] protect the program against COOP attack by guaranteeing that the callee of every call site is legitimate. Both function pointers and virtual function calls are vulnerable indirect function calls inducing forward control-flow transfer that may be exploited to launch a code reuse attack. There are plenty works that study the protection of virtual calls in both binary level, such as VTPin [34], vfGuard [35], VTint [36], and TypeArmor [28], and source code level, such as Shrinkwrap [37] and VTrust [27]. But there still exist problems in current work about protecting function

pointers, because they may be bypassed when facing dynamic code reuse attack on forward edges of CFG, as analyzed in the work Newton [16].

There are several existing CFI-based defense methods that focus on protecting forward edges of CFG, such as Bin-CFI [13], FSan [8], MCFI [10], and TypeArmor [28]. They construct the legitimate target sets for every indirect function calls and enforce transfers of indirect function calls to the corresponding legitimate target sets. Bin-CFI [13] protects the forward-edge integrity by constraining indirect function calls to the legitimate target set of all function entry address. TypeArmor [28] constructs the legitimate target set by checking whether the parameter number of target function is not more than the argument number of the corresponding indirect call site at binary level. FSan [8] and MCFI [10] construct the legitimate target set by making sure the type of the target function is the same as the type of the corresponding indirect call site.

However, a new code reuse attack system called Newton [16] that is based on dynamic analysis and aims at attacking forward-edge control-flow equipped with advanced techniques may also bypass the above defense methods.

In this paper, we propose a fine-grained CFI-based defense method called P-CFI to mitigate it. P-CFI constructs the legitimate target sets of indirect function calls based on *points-to* analysis of call site value and function type information.

3. Threat Model

In our threat model, we assume that the target program is written with memory unsafe programming language, which means that the attacker can corrupt the memory of it via memory-related vulnerabilities, such as buffer overflow, inside the target program. The DEP defense is in place in the environment where the target program is running on while the ASLR defense is turned off. We also assume that the DEP cannot be bypassed using memory remap or any other attack techniques. The attacker can read and write the data pages in memory, and he is also able to read and execute instructions in code pages. We consider the attacker can take control of function pointers and can compromise it with any value using dynamic analysis and taint analysis methods implemented in the tool of the work Newton [16].

As for a defense method, we protect the transfers of indirect function calls which are analyzed from the source code of the target program. Legacy binaries with no debug information and no source code are not considered in our work. We instrument additional checking instructions in the process of generating executable binaries to restrain the forward-edge control-flow from transferring to undesired addresses and therefore defend the program from compromised function pointers and ensure forward-edge control-flow integrity.

4. Method Overview

A function pointer is a variable that points to the start address of a function. This is a kind of indirect function call and can

be compromised by attackers if the target programs include memory-related vulnerabilities. As mentioned before, the technique based on the dynamic analysis and taint analysis can combine tainted values and function pointers to launch a successful code reuse attack. The legitimate target set of the indirect call site which is constructed by recent defense methods is too coarse. For instance, Bin-CFI [13] and CCFIR [9] regard all function addresses as the legitimate target set for every indirect call site. IFCC [8] classifies the legitimate target set for every indirect call site according to the numbers of function parameters. FSan [8] constructs the legitimate target set by mapping call site types to target function types. CsCFI [38] tracks paths to sensitive program states and defines the set of valid control edges within the state context. CPI [14] protects the sensitive code pointer by putting them in hidden memory location. In our work, we refine the legitimate target set for each function pointer at each indirect call site to ensure the precision of forward-edge CFI. This requires us to distinguish the legitimate function address from illegitimate one before transferring the control-flow to the destination of an indirect function-call. We also need to develop a mechanism for the target program to instrument the CFI check for each indirect call site. To this end, we propose a fine-grained CFI-based defense method for forward edges of CFG based on *points-to* analysis to mitigate dynamic code reuse attack caused by the work Newton [16]. We achieve this by protecting the indirect function call and forcing the program to check the transfer of the indirect function call before calling it. We also develop a prototype for the proposed method named P-CFI. P-CFI contains two phases: static analysis and policy enforcement.

The first phase of P-CFI is shown in Figure 1. We translate the source code to LLVM intermediate representation (LLVM IR) with debug information and analyze it to find out all function pointers that need to be protected. We also need to find out the set of reachable function entries for every function pointer by using *points-to* analysis [17]. Then, we find out the indirect call sites corresponding to these function pointers by using *def-use* analysis. The reachable function entries of the function pointer is the legitimate target of the indirect function call whose call value is this function pointer.

The second phase of P-CFI is shown in Figure 2. We construct function entry tables in the link-time optimizing process of compiling source code according to the legitimate target set obtained from previous phase. There is only one function entry table for each function pointer at different indirect call sites. We enforce the security policy by adding checking instructions before each indirect call site and forcing them to check the values according to the pregenerated tables to ensure that illegitimate transfers will be trapped.

It has to be noted that our method is more precise than FSan [8] and TypeArmor [28]. FSan [8] ensures that the type of the target function is the same as the type of the corresponding indirect call site. TypeArmor [28] ensures that the parameter numbers of the target function are equal to or less than the argument numbers of the corresponding indirect call site.

FIGURE 1: Work flow of static analysis.

FIGURE 2: Work flow of policy enforcement.

5. Design of P-CFI

The process of P-CFI can be divided into two steps, static analysis and policy enforcement. We will introduce the details of the concept and design in this section, respectively.

We will use the following two code pieces to explain the design of the proposed method. Listing 1 gives an example of declarations of functions and function pointers in a C program, which includes three functions with no arguments and three function pointers with no arguments. Listing 2 gives an example that illustrates the process of function address assignment and indirect function calls in C program.

5.1. Static Analysis. In this phase, we need to identify all function entries and each entry belongs to one or more function pointers. The declaration of a function pointer and the locations of indirect call sites according to this function pointer should be identified in this phase as well. We also need to find out the correspondence between target function entries and the indirect call site. We translate the source code of target program into LLVM intermediate representation (LLVM IR) with debug information first for easing the analysis. P-CFI is different from previous CFI methods, because we analyze the set of reachable function entries for every function pointer by using *points-to* analysis [17] and the relationship between indirect call sites and function pointers to obtain the legitimate target set for every indirect function calls.

Function Entry Analysis. The functions that have their addresses assigned to function pointers are known as *address-taken* functions. They are the basic elements of function entry tables. We can obtain every *address-taken* function through analyzing LLVM IR. As shown in Listing 1, all functions that have been assigned to function pointers will be found at first. The function `foo()` and function `bar()` declared in line 1 and line 2 are identified as function entries of indirect function calls, while the function `cat()` is ignored for that

its address has never been assigned to any function pointer in Listing 2.

Function Pointer Analysis. The function pointer analyzer scans LLVM IR of the target program to identify the function entry set for every function pointer that requires protection. We do the *points-to* analysis [17] for every function pointer and get the reachable function entries for every function pointer. Then, we get the function entry set for the function pointer by deleting the function entries which are not the *address-taken* function. As shown in Listing 2, the function entry set of the function pointer `pointer_one` contains the function `foo()` and the function `bar()`.

Indirect Call Site Analysis. The term “indirect call site” refers to the location of code where control-flow transfer caused by function pointers happens. We need to find all locations in the source code that invoke these function pointers, because these are the very positions where policy enforcement will be deployed. Usually, the control-flow transfer caused by the invocation of function pointers is not taken into consideration in the CFG generated in the process of compiling, because it is a kind of indirect function call and its destination can only be dynamically decided at runtime.

To address this issue, we construct the legitimate target set for every indirect call site. We use *def-use* analysis to get the corresponding function pointer of the indirect call site. The function entry set of the function pointer is the legitimate target of the indirect function call whose call value is this function pointer. As shown in Listing 2, the legitimate target set of the indirect function call site `pointer_one()` contains the function `foo()` and the function `bar()`.

FSan [8] uses the function type to construct the legitimate target set for the indirect call site. It means that the indirect function call site `pointer_one()` has the legitimate target set which contains the functions `foo()`, `bar()`, and `cat()`. However, in our method, the indirect function call site `pointer_one()` can only call the functions `foo` and `bar()`. To construct more precise legitimate target set, we generate

FIGURE 3: The call graph of sample code in Listing 2.

the legitimate target sets for every indirect function calls by using *points-to* analysis.

The comparison of the call graph of code in Listing 1 for P-CFI and FSAN is illustrated in Figure 3. As shown in Figure 3(a), the legitimate target set of the indirect function call cite `pointer_one()` contains functions `foo` and `bar`, the legitimate target set of the indirect function call cite `pointer_two()` also contains functions `foo` and `bar`, and the legitimate target set of the indirect function call cite `pointer_three()` only contains function `foo`. As shown in Figure 3(b), the legitimate target set of all the three indirect function call cites contains the same functions `foo`, `bar`, and `cat`. We conclude that the call graph of P-CFI is more precise than that of FSAN.

Virtual Call Analysis. Virtual calls are common indirect calls in C++. Once a call to a virtual function of an object is invoked, the virtual table pointer is dereferenced to locate the virtual table. Then, the corresponding virtual function pointer is located by using the offset of this virtual function in the virtual table. Finally, this pointer is used to invoke the virtual call. By exploiting vulnerabilities in the program, attackers can modify the virtual table pointer and point it to a bogus virtual table that they have made. With this method, attackers' code can be executed through the incorrect virtual table pointer. Therefore, the virtual table pointer is the specific target by attackers to subvert a forward edge in the control-flow graph.

We need to do the *points-to* analysis for every virtual table pointer and get the valid virtual table for it. Although LLVM bitcode is very suitable for representing C, it is not very suitable for C++, because high-level features are translated to low-level constructs. For example, virtual tables are translated as arrays of function pointers, and virtual calls are translated to normal indirect calls. The work in [18] proposes a *points-to* analysis method, which offers a structure-sensitive analysis that can recover much of the available high-level structure information of types and objects for C++. We use this method to only analyze the virtual calls.

5.2. Policy Enforcement. This policy enforcement consists of static instrumentation in LLVM IR and runtime check. The process of static instrumentation instruments check logic before indirect call cite and construct legitimate targets set for every indirect function call. The runtime check process

```

1 void foo () {return;}
2 void bar () {return;}
3 void cat () {return;}
4
5 void (* pointer_one)();
6 void (* pointer_two)();
7 void (* pointer_three)();

```

LISTING 1: Code snippet of function declarations and pointer declarations.

```

1 int main(int argc, char ** argv) {
2
3     int num;
4     scanf("%d", &num);
5
6     if(num == 1) {
7         pointer_one = foo;
8     } else {
9         pointer_one = bar;
10    }
11    pointer_two = foo;
12    pointer_two = bar;
13    pointer_three = foo;
14
15    pointer_one();
16    pointer_two();
17    pointer_three();
18
19    return 0;
20 }

```

LISTING 2: Code snippet of function pointer assignments and indirect call cites.

ensures that the function entry address corresponds to indirect call cite in the legitimate target set.

Static Instrumentation. The implementation of the static instrumentation is based on `llvm.bitset.test` intrinsic and `llvm.bitsets` global metadata in LLVM IR provided by LLVM version 3.8. We use the function entry table to represent legitimate function entry for each indirect call cite

```

1 pointer_one_FET:
2   jmp foo;
3   jmp bar;
4 pointer_two_FET:
5   jmp foo;
6   jmp bar;
7 pointer_three_FET:
8   jmp foo;

```

LISTING 3: Function jump table of indirect function calls.

and force every call cite to jump to this function entry table and compare their destination function entry address with the values in function entry table. The `llvm.bitset.test` intrinsic can be used to test whether the value of a given pointer is an element in given bitset. It is used to check whether the callee of indirect call cite is in the legitimate function entry table.

A function entry table contains all legitimate function addresses that a pointer in a certain indirect call site may point to. For example, the legitimate target set of the indirect call site `pointer_one()` in Listing 2 line 15 contains the functions `foo()` and `bar()`. As a result, the entry address of function `foo()` and function `bar()` will be put into the function entry table of this indirect call cite. The entry table of the indirect function call cite `pointer_two()` in line 16 of Listing 2 contains the addresses of the functions `foo()` and `bar()`, while the the entry table of the indirect function call cite `pointer_three()` in line 17 of Listing 2 only consists of the address of the function `foo()`. The function entry tables of them are shown in Listing 3.

The function entry table will be integrated in `llvm.bitsets` global metadata. The LLVM backend will compile the instrumented LLVM IR to execute file.

Runtime Check. We use the `llvm.bitset.test` intrinsic and `llvm.bitsets` global metadata of LLVM IR as our instrument tools. According to the semantics of `llvm.bitset.test`, the instrumented check logic will check the callee value of indirect call cite in the legitimate target set.

6. Implementation

Our method is implemented on Ubuntu16.04.1 x86_64 with C++. The development of our prototype is based on LLVM with version 3.8. There are two components in our implementation. The component of static analyzer is responsible for finding function pointers, indirect call cites, and virtual table points corresponding to legitimate target sets. The other component is responsible for constructing function entry table and instrumenting check logic at each indirect call cite, or constructing virtual table pointer set and instrumenting check logic at each virtual table pointer dereferencing place. Our implemented prototype supports target programs written in C and C++.

We insert the `llvm.bitset` metadata into the LLVM bitcode and insert `llvm.bitset.test` check before indirect calls. For C language programs, taking the indirect function call of line 16 in Listing 1 as an example, before the `bitsets` metadata and indirect function call checks are inserted, the LLVM bitcode corresponding to this call is shown in Listing 4. The function pointer `%4` is assigned the address of the function `bar` before the indirect function call is made. But at this moment, there is no check on the value of function pointer `%4`, so an attacker can modify this function pointer to execute arbitrary code.

After the `bitsets` metadata and indirect function call checks are inserted, the LLVM bitcode corresponding to its call is shown in Listing 5. Lines 16 and 17 are the `bitsets` set of the objective functions corresponding to the check code inserted in line 5, which contains the target functions `foo` and `bar`. Line 4 checks the function pointer by checking whether the address of the function pointer is in the `bitsets` set corresponding to `_ZTSFvE.test.c.16`. If yes, the program jumps to line 11 and then executes code in line 12 to call indirect function. Otherwise, the program jumps to line 7 and then executes code in line 8 to terminate the execution.

Listing 6 presents a simplified version of the LLVM bitcode for the translation of a virtual call in C++ object into normal indirect call. More precisely, the code snippet shows the translation of virtual call `tp->foo()` for object `Test *tp`.

As discussed in Section 5.1, we perform the *points-to* analysis by using the method proposed by [18] to analyze the virtual table pointer for C++. With this method, both `\ %tp` and `\ %1` (after the cast) will point both to the stack-allocated `Test` object and to its virtual call pointer field. The first load instruction will return the virtual call table. Indexing into the virtual table will return the array element for the subobject. Then, the second load instruction will return the exact function that the virtual table points to at the given offset. Finally, the virtual call will be made in the last line.

We insert the `llvm.bitset` metadata into the LLVM bitcode to constrain the set of valid virtual table pointers for the call site and insert `llvm.bitset.test` check after the cast of virtual table pointer (between line 2 and line 3 of Listing 6) to verify whether the virtual table pointer from the object is in the valid set. If yes, the virtual table pointer will then be used to obtain the virtual table. Otherwise, an error will be reported and the execution of the program will be terminated.

7. Evaluation

We run our evaluation on the Ubuntu16.04.1 with kernel version 4.13.0. The physical machine is equipped with Intel Core i5-4590 3.3GHz processor and 6GB memory. The dataset of our evaluation consists of seven programs of SPEC CPU2006 suites. The performance and functionality are evaluated, respectively, using fore-mentioned testbed and dataset.

```

1 store void (...) * bitcast (void () * @bar to void
  (...) *), void (...) ** %2, align 8 // Assign
  the address of function bar to %2
2 %4 = load void (...) * , void (...) ** %2, align 8 //
  Assign %2 to %4
3 call void (...) %4() // Function call by calling
  the function corresponding to the %4 address

```

LISTING 4: LLVM bitcode before instrumentation.

```

1 store void (...) * bitcast (void () * @func2 to void
  (...) *), void (...) ** %2, align 8 // Assign the
  address of the function bar to %2
2 %4 = load void (...) * , void (...) ** %2, align 8 //
  Assign %2 to %4
3 %5 = bitcast void (...) * %4 to i8 *, !nosanitize !3
4 %6 = call i1 @llvm.bitset.test(i8 * %5, metadata !"
  _ZTSFvE_test.c.13"), !nosanitize !3 // Determine
  if %5 is in the bitset set corresponding to
  _ZTSFvE_test.c.13
5 br i1 %6, label %8, label %7, !nosanitize !3
6
7; <label>:7:    ; preds = %0
8 call void @llvm.trap() #2, !nosanitize !3
9 unreachable, !nosanitize !3
10
11; <label>:8:    ; preds = %0
12 call void (...) %4()
13
14 !llvm.bitsets = ! { !0, !1}
15
16 !0 = ! { !" _ZTSFvE_test.c.16", void () * @foo, i64 0}
17 !1 = ! { !" _ZTSFvE_test.c.16", void () * @bar, i64 0}

```

LISTING 5: LLVM bitcode after instrumentation.

```

1 %class.Test = type { i64 (...) **, ...}
2 %1=bitcast %tp to i64 (%class.Test *) ***
3 %2=load i64 (%class.Test *) ** %1
4 %3=getelementptr i64 (%class.Test *) ** %2, 1
5 %4=load i64 (%class.Test *) * %3
6 call i64 %4 (%class.Test * %bp)

```

LISTING 6: LLVM bitcode for a C++ virtual call.

7.1. Performance Evaluation. We evaluate the performance of P-CFI and analyze the result briefly. To measure the runtime overhead, we use SPEC CPU2006 benchmarks to test the time overhead of P-CFI, and we normalize the result against the baseline and compare the result with FSAN which is presented in forward-edge CFI [8]. Figure 4 illustrates the results.

We measure the time to complete the execution of programs which have indirect function calls in benchmark and normalize the result with baseline obtained by executing

original programs in benchmark with no instrumentation. FSAN [8] represents the CFI method considering type information of indirect call site and function. The legitimate target set of P-CFI is more precise than FSAN, and they have the similar overhead. As shown in Figure 4, for the seven C benchmarks on the left side, the relative overhead introduced by deploying our enforcement method is very low, with the average value of 2.04% and worst case 3.47%. For the four C++ benchmarks on the right side, the relative overhead

TABLE 1: Comparison of numbers of functions in legitimate target sets.

Defend methods	max	min	median
TypeArmor(binary)	721	228	325
IFCC(type src)	267	2	189
FSan(src)	198	1	9
P-CFI(src)	110	1	5

FIGURE 4: Comparison of performance overhead between P-CFI and FSan using SPEC CPU2006 benchmarks.

introduced is also acceptable, with the average value of 6.01% and worst case 9.54%. Overall, the impact of time overhead introduced by P-CFI is relatively trivial in every program considered in SPEC CPU2006 benchmarks and is acceptable in real world situation.

7.2. Security Evaluation. We prove the effectiveness of P-CFI by analyzing how P-CFI can defend against dynamic code reuse attack caused by the work *Newton* [16] by using a case study and compare our statistical result with existing defense techniques to illustrate the security guarantee provided by P-CFI. We use the source code of *nginx* with the version 1.3.9 as the target program and compile it with P-CFI. An example of dynamic code reuse attack illustrated in *Newton* [16] let a function pointer inside a program have the same type with function `malloc` in `libc` point to `malloc` followed by `mprotect` to make the code page of `libc` writable. This attack can bypass the protecting method based on context-sensitive CFI, such as *PathArmor* [38]. As for P-CFI, since the function address of `malloc` not appeared in the function entry table of that function pointer, the transfer to `malloc` will not be allowed by P-CFI's runtime check.

P-CFI can lower the function numbers of the legitimate target set for each indirect call cite. We compute and compare the size of the legitimate target set for each indirect call cite

with TypeArmor [28], FSan [8], and IFCC [8] to illustrate the effectiveness. The results are listed in Table 1. The maximal size of the legitimate target set, the minimal number of the legitimate target set, and the median number of the legitimate target set, for all indirect call cites with different defense technique, are taken into consideration in this comparison.

The result in Table 1 shows that P-CFI can lower the size of the legitimate target set for each indirect function call and ensure that 50% (median value) of the legitimate target sets are with size less than 5, which means that the functions that indirect function calls can jump to are less in P-CFI than that in other methods. In conclusion, P-CFI can protect the forward control-flow in a more precise way with more solid security guarantee.

8. Conclusion

In this paper, we propose a fine-grained CFI named P-CFI for forward edges of CFG based on *points-to* analysis to defend CPS against dynamic code reuse attacks enabled by the work *Newton* [16]. P-CFI analyzes target program in source code level and protects the function pointer to ensure the regular flow of control will not be violated. Points-to analysis is employed to generate precise legitimate target sets for each indirect call cite corresponding to a function pointer. The implementation can output instrumented binaries that force every indirect function call caused by function pointer jump to a function entry table. The security analysis also proves the P-CFI can constrain the range of reusable code pieces and lower the probability of successful code reuse attack. Performance evaluation shows that the overhead imposed by P-CFI is negligible.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China under grant No. 61772221, in part by the Shenzhen Fundamental Research Program under grant No. JCYJ20170413114215614, and in part by the Fundamental

Research Funds for the Central Universities under grant No. HUST2018KFYYXJJ049.

References

- [1] Y. Ding, T. Wei, H. Xue, Y. Zhang, C. Zhang, and X. Han, "Accurate and efficient exploit capture and classification," *Science China Information Sciences*, vol. 60, no. 5, 2017.
- [2] Microsoft, "Data Execution Prevention (DEP)," <http://support.microsoft.com/kb/875352/EN-US/>, 2006.
- [3] C. Cowan, C. Pu, D. Maier et al., "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium*, pp. 63–78, 1998.
- [4] E. D. Berger and B. G. Zorn, "DieHard: probabilistic memory safety for unsafe languages," in *Proceedings of the the 2006 ACM SIGPLAN conference*, p. 158, Ottawa, Ontario, Canada, June 2006.
- [5] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, pp. 121–141, 2011.
- [6] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, pp. 552–561, ACM, November 2007.
- [7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pp. 340–353, November 2005.
- [8] C. Tice, T. Roeder, P. Collingbourne et al., "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proceedings of the the 23rd USENIX Security Symposium*, pp. 941–955, 2014.
- [9] C. Zhang, T. Wei, Z. Chen et al., "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP) Conference*, pp. 559–573, Berkeley, CA, USA, May 2013.
- [10] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the the 35th ACM SIGPLAN Conference*, pp. 577–587, Edinburgh, United Kingdom, June 2014.
- [11] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Proceedings of the 35th IEEE Symposium on Security and Privacy, SP 2014*, pp. 292–307, USA, May 2014.
- [12] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, and L. Szekeres, "Protecting function pointers in binary," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS 2013*, pp. 487–492, China, May 2013.
- [13] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the 22nd USENIX Security Symposium*, pp. 337–352, 2013.
- [14] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pp. 147–163, 2014.
- [15] PaX-team, "PaX ASLR (Address Space Layout Randomization)," <http://pax.grsecurity.net/docs/aslr.txt>, <http://pax.grsecurity.net/docs/aslr.txt>, 2003.
- [16] V. van der Veen, D. Andriess, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida, "The Dynamics of Innocent Flesh on the Bone," in *Proceedings of the the 2017 ACM SIGSAC Conference*, pp. 1675–1689, Dallas, Texas, USA, October 2017.
- [17] L. O. Andersen, *Program Analysis and Specialization for the C Programming Language [Ph.D. dissertation]*, University of Copenhagen, 1994.
- [18] G. Balatsouras and Y. Smaragdakis, "structure-sensitive points-to analysis for C and C++," in *Proceedings of 2016 International Static Analysis Symposium*, pp. 84–104, 2016.
- [19] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: the next computing revolution," in *Proceedings of the Proceeding of the 47th Design Automation Conference (DAC '10)*, pp. 731–736, New York, NY, USA, June 2010.
- [20] A. Humayed, J. Lin, F. Li, and B. Luo, "Cyber-physical systems security - a survey," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1802–1831, 2017.
- [21] E. G. Chekole, J. H. Castellanos, M. Ochoa, and D. K. Yau, "Enforcing memory safety in cyber-physical systems," in *Computer Security*, vol. 10683 of *Lecture Notes in Computer Science*, pp. 127–144, Springer International Publishing, Cham, 2018.
- [22] N. Carlini and D. Wagner, "ROP is still dangerous: breaking modern defenses," in *Proceedings of the 23rd USENIX Security Symposium*, pp. 385–399, 2014.
- [23] L. Davi, A. R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection," in *Proceedings of the 23rd USENIX Security Symposium*, pp. 401–416, 2014.
- [24] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in C++ applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy, (SP '15)*, pp. 745–762, May 2015.
- [25] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the the 10th ACM Symposium*, pp. 555–566, Singapore, Republic of Singapore, April 2015.
- [26] S. J. Crane, M. Franz, S. Volckaert et al., "It's a TRaP," in *Proceedings of the the 22nd ACM SIGSAC Conference*, pp. 243–255, Denver, Colorado, USA, October 2015.
- [27] C. Zhang, S. A. Carr, T. Li et al., "VTrust: regaining trust on virtual calls," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, 2016.
- [28] V. V. D. Veen, E. Goktas, M. Contag et al., "A tough call: mitigating advanced code-reuse attacks at the binary level," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy, SP 2016*, pp. 934–953, USA, May 2016.
- [29] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the the 22nd ACM SIGSAC Conference*, pp. 268–279, Denver, Co, USA, October 2015.
- [30] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, (CCS '15)*, pp. 941–951, ACM, Denver, Co, USA, October 2015.
- [31] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: overcoming control-flow integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*, pp. 575–589, San Jose, CA, USA, May 2014.
- [32] I. Evans, F. Long, U. Otgonbaatar et al., "Control Jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer*

- and Communications Security, CCS 2015*, pp. 901–913, USA, October 2015.
- [33] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “Control-flow bending: on the effectiveness of control-flow integrity,” in *Proceedings of the 24th USENIX Security Symposium*, pp. 161–176, 2015.
 - [34] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos, “VTPin: practical VTable hijacking protection for binaries,” in *Proceedings of the 32nd Annual Conference*, pp. 448–459, Los Angeles, Calif, USA, December 2016.
 - [35] A. Prakash, X. Hu, and H. Yin, “vfGuard: strict protection for virtual function calls in COTS C++ binaries,” in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, 2015.
 - [36] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, “VTint: protecting virtual function tables’ integrity,” in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, 2015.
 - [37] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos, “ShrinkWrap: vtable protection without loose ends,” in *Proceedings of the 31st Annual Computer Security Applications Conference*, pp. 341–350, Los Angeles, CA, USA, December 2015.
 - [38] V. van der Veen, D. Andriesse, E. Göktaş et al., “Practical Context-Sensitive CFI,” in *Proceedings of the 22nd ACM SIGSAC Conference*, pp. 927–940, Denver, Co, USA, October 2015.