FINE-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels

Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma

***Abstract*— The operating system kernel is often the security foundation for the whole system. To prevent attacks, control-flow integrity (CFI) has been proposed to ensure that any control transfer during the program's execution never deviates from its control-flow graph (CFG). Existing CFI solutions either work in user space or are coarse-grained; thus they cannot be readily deployed in kernels or are vulnerable to state-of-the-art attacks. In this paper, we present FINE-CFI, a system that enforces fine-grained CFI for operating system kernels. Unlike previous systems, FINE-CFI constructs the kernel's fine-grained CFG with a retrofitted context-sensitive and field-sensitive pointer analysis, then enforces CFI with this CFG. At the same time, FINE-CFI provides comprehensive protection to the control data in the kernel's interrupt context. Combining the above two kinds of protection, we can thus defeat those formidable ret2usr and kernel code-reuse attacks. We have developed a compiler-based prototype and implemented this technique in Linux 3.14 kernel. Our evaluation indicates that FINE-CFI prevents all the gadgets found by an open-source gadget-finding tool from being misused, as well as all the attacks from the RIPE benchmark and malicious attempts to modify control data in the interrupt context; and it also reduces the number of indirect control-flow targets by 99.998%, thus largely raising the bar for attackers. Our evaluation also shows that the performance overhead introduced by FINE-CFI is less than 10% on average.**

***Index Terms*— Control-flow integrity, kernel protection, fine-grained, intrusion prevention.**

I. INTRODUCTION

MOST commodity operating systems are written in unsafe languages, e.g., C/C++, hence are vulnerable to memory safety attacks. Specifically, attackers can exploit a buffer overflow vulnerability to overwrite a function pointer or return address data in memory to redirect the program execution to anywhere they want.

As operating system kernel is often the security foundation for the whole system, researchers have proposed a number of solutions to counter memory safety attacks. For instance, DEP (Data Execution Prevention) [1], or W⊕X (Write XOR Execute) [2], prevents code injection attacks by marking a memory page as writable or executable, but not both at the same time. That is to say, it blocks attacks that inject malicious code as data while later execute it as code. Unfortunately, such a guarantee cannot defend against code-reuse attacks, e.g., return-into-libc [3], return-oriented programming [4]–[7], or jump-oriented programming [8]. These code-reuse attacks can be performed by only misusing legitimate code snippets (or "gadgets") and chaining them together with some special instructions, e.g., *ret* or *indirect jmp*. Researchers have demonstrated that these code-reuse attacks can achieve Turing-complete computations, thus can do anything they want. To defeat code-reuse attacks, ASLR (Address Space Layout Randomization) [9] places code and data segments at random addresses, making it harder for attackers to reuse existing code for execution. Unfortunately, ASLR can be bypassed via information leaks, or timing side channel attacks [10].

From another perspective, most memory safety attacks work by hijacking a program's control flow to redirect its execution [11]. Control-Flow Integrity (CFI) [12] has been proposed to thwart such attacks. CFI ensures that a program's control flow follows a statically computed Control-Flow Graph (CFG). It inserts inline reference monitor code into the program and enforces safety check during execution. When the program deviates its CFG, its execution is stopped and often an alarm is raised or logged. As CFI is resistant to code injection, code-reuse, and information leakage attacks, it has been widely studied for more than ten years.

An array of user-level CFI systems have been designed, and most of them are coarse-grained [12]–[15]. For instance, Bin-CFI [13] uses two tags (or labels) to indicate the valid control transfer targets. One tag is for all the indirect call/jmp sites, the other is for all the return sites. CCFIR [14] leverages three tags, one for all the indirect call/jmp sites, one for return sites of "non-sensitive" functions, and a special one for return sites of sensitive functions (e.g., *system()* function that can be used to execute a file or create a process). FECFI [15] classifies function pointers by the number of arguments for C code (leading to an effective limit around eight) or into a single class. Unfortunately, researchers have demonstrated that all these coarse-grained CFI systems can be bypassed [16]–[19].

As a result of the attacks on coarse-grained CFI systems, several fine-grained CFI solutions have been proposed [20]–[22]. For instance, CCFI [22] enforces fine-grained CFI by computing and storing a message

Manuscript received July 7, 2017; revised November 23, 2017 and January 9, 2018; accepted January 13, 2018. Date of publication January 25, 2018; date of current version February 12, 2018. This work was supported in part by the Natural Science Basic Research Plan in Shaanxi Province of China under Grant 2015JM6351, in part by the Key Program of the National Natural Science Foundation of China under Grant U1405255, and in part by the Shaanxi Science and Technology Coordination and Innovation Project under Grant 2016TZC-G-6-3. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Mauro Conti. *(Corresponding author: Jinku Li.)*

J. Li, X. Tong, and J. Ma are with the School of Cyber Engineering, Xidian University, Xi'an 710071, China (e-mail: jkli@xidian.edu.cn; txmxidian@gmail.com; jfma@mail.xidian.edu.cn).

F. Zhang is with the Department of Computer Science, Wayne State University, Detroit, MI 48202 USA (e-mail: fengwei@wayne.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2018.2797932

authentication code of control flow objects each time they are stored in memory, and checking every time the value is loaded from memory. By doing so, CCFI prevents the attacker from writing an arbitrary address to hijack execution. CPI [20] instead provides protection by identifying and collecting all the program's sensitive pointers into a safe region, and preventing all the normal code from accessing this region. Although researchers have demonstrated CPI's ineffectiveness on x86-64 and ARM architecture [23], CPI's authors argue that their implementation alternatives using hardware-enforced segmentation (e.g., on x86-32) or software fault isolation cannot be subverted [24]. Note that these fine-grained invariants of CFI may be immune to those attacks targeting coarse-grained CFI systems, but all these invariants in user level do not provide protection for the control data in kernels, thus are vulnerable to ret2usr attacks. It is worth clarifying that a classic ret2usr attack [25] only corrupts non-interrupt control data (e.g., a kernel function pointer or return address), however, the novel ret2usr attacks could corrupt CS and IP in the interrupt context, to transfer execution control-flow to the user-space code controlled by attackers with kernel privileges. As a result, it is also important to protect control data in the interrupt context.

On the other hand, additional challenges are present to enforce kernel's control-flow integrity. The first challenge is due to the asynchronous nature of context switching and interrupt handling in kernels. In particular, when an interrupt occurs, the interrupted program's context information is saved to memory for later restoring when the program is resumed. Note that the saved context information could be possibly tampered with by attackers to hijack the control flow. Different from the control-flow transfer with an indirect call/jmp or ret instruction, we cannot predetermine when and where an interrupt may occur, as interrupts could occur at any time or any valid instruction boundary. The second challenge comes from the heavy use of single code pointers in various structures in kernel code, and these pointers may scatter across the whole kernel. To enforce fine-grained CFI for the kernel, it is required to identify all these code pointers and leverage them to infer the point-to set for each indirect function call, which is not an easy task to implement. The third challenge is that the performance overhead introduced by the enforcement should be as low as possible, as the operating system kernel is the foundation for all the applications.

Despite the challenges, several kernel-level CFI systems have been presented by researchers [26]–[28]. For instance, HyperSafe [26] enforces CFI protection for a tiny hypervisor (kernel). However, HyperSafe does not provide protection for control data in the interrupt context, thus is vulnerable to ret2usr attacks. In contrast, KCoFI [27] provides complete control-flow integrity for commodity operating system kernels. KCoFI protects not only the transfer targets of all the *indirect call/jmp* and *ret* instructions, but also the control data (e.g., return address) in context switching, interrupt handling, and signal handler dispatch, thus is immune to ret2usr and kernel code-reuse attacks. However, KCoFI incurs a high performance overhead (over 100%), which slows down the whole system including all the applications on it. What's worse, to escape from sophisticated whole-program analysis or complete memory safety enforcement, KCoFI only provides coarse-grained CFI for the indirect call/jmp and return sites. Specifically, KCoFI leverages only one label for the targets of all the indirect call/jmp and return sites. As mentioned earlier, the coarse-grained CFI systems can be subverted by attacks chained with new gadgets [16]–[19]. Compared with KCoFI, *indexed hooks* [28] enforces two different-grained invariants of CFI, i.e., coarse-grained CFI and fine-grained CFI. Specifically, for the fine-grained CFI, *indexed hooks* sets up multiple jump tables with the input of a fine-grained CFG, and each table contains the legal target addresses for a specific *indirect call/jmp* or *ret* instruction. When an indirect transfer occurs, *indexed hooks* ensures that its target address is one item in its jump table. To obtain the fine-grained CFG, *indexed hooks* leverages dynamic analysis, which runs a FreeBSD virtual machine on QEMU [29] to profile the targets of indirect calls. Note that dynamic analysis has an incomplete coverage as it is impossible to reach all the indirect calls. The experiments show that only 42.67% indirect calls are reached in its profiling. For those unreachable indirect calls, *indexed hooks* conservatively assumes a maximum set, which contains the addresses of all the functions that may be invoked. This assumption could lead to a coarse-grained CFI, which has been demonstrated vulnerable by researchers. Thus, we argue that we need a lightweight and real fine-grained CFI for operating system kernels, which can not only provide fine-grained CFI for all the indirect control transfer instructions (i.e., *indirect call/jmp*, and *ret*), but also protect control data in the interrupt context, to defeat the ret2usr and kernel code-reuse attacks.

In this work, we present FINE-CFI, which enforces fine-grained control-flow integrity for operating system kernels. Unlike previous systems, FINE-CFI constructs kernel's fine-grained CFG with a retrofitted context-sensitive and field-sensitive pointer analysis, then enforces CFI with this CFG. As well, FINE-CFI provides protection to the control data in kernel's interrupt context. With both of them, FINE-CFI can effectively defeat those formidable ret2usr and kernel code-reuse attacks.

To enforce fine-grained CFI for a kernel, we need a fine-grained CFG of the kernel. To obtain the fine-grained CFG, the key is to get the point-to set for every and each *indirect call/jmp* instruction. To address this problem, we propose a retrofitted point-to analysis approach. The main innovation of our approach is that we introduce a new vector, called *struct location vector*, to infer the targets for indirect function calls. We introduce such a vector as we observe that there are a large number of function pointer initializations and assignments inside struct variables in kernel space, and the function pointer locates in the same field of the struct in its lifetime, i.e., from the initialization or assignment to be consumed by indirect function calls. As a result, it largely reduces the number of targets of *indirect call/jmp* instructions and improves the precision of results.

While for the protection of control data in the interrupt context, the static analysis approach is not applicable as we cannot predetermine when and where an interrupt may occur. Consequently, we propose a hypervisor-based approach, which backups the runtime context information (e.g., the CS and IP) in the hypervisor when an interrupt arrives, and then verifies that information by comparing the values in current kernel stack and in the hypervisor when the interrupt returns. By doing so, we enforce the protection for control data in the interrupt context.

To validate our approach, we have developed a proof-of-concept prototype with the open-source LLVM compiler [30] and implemented this technique in Linux 3.14/x86-amd64 kernel. As a compiler-based approach, we need to recompile the protected OS kernel source code and enforce protection for it. Then, we conduct a number of attacking and benchmark-based experiments on the system, and the results demonstrate the effectiveness and efficiency of our approach.

In summary, our paper makes the following contributions:

- We propose a retrofitted static analysis approach, and leverage it to obtain the point-to sets of *indirect call/jmp* instructions. Our approach largely reduces the number of targets of *indirect call/jmp* instructions (average 13.14 for each *indirect call/jmp* instruction). Further, we construct the fine-grained CFG of the kernel and use it to enforce fine-grained CFI for the kernel.
- We propose a hypervisor-based approach to provide protection for control data in the interrupt context. Combining the above two kinds of protection, we thus can defeat the ret2usr and kernel code-reuse attacks.
- To validate our approach, we have developed a compiler-based prototype, called FINE-CFI. In particular, we leverage LLVM compiler to enforce protection for Linux 3.14/x86-amd64 kernel. Then we perform a systematic security analysis and a number of attacks to FINE-CFI. The results indicate that FINE-CFI prevents all the attacks and it reduces the number of indirect control-flow targets by 99.998%, thus largely raises the bar for attackers. We also perform several benchmark-based performance measurements, the results show that the performance overhead introduced by our system is less than 10% on average with fine-grained CFI for all *indirect call/jmp* and *ret* instructions, as well as the protection of control data in the interrupt context.

The rest of the paper is organized as follows. First we describe the threat model and assumptions, and present the key techniques in Section II. Then we show the implementation details and evaluation results in Sections III and IV, respectively. After that, we discuss possible limitations of our current prototype in Section V and describe related work in Section VI. Finally, we conclude our paper in Section VII.

II. SYSTEM DESIGN

A. Overview

1) Threat Model and Assumptions: In this work, we assume that attackers can obtain the highest privilege inside the OS (e.g., the *root* privilege in Linux) and full access to the system memory space. At the same time, we assume that attackers can overwrite the control data (e.g., function pointers or return addresses) in kernel memory to deviate kernel's control flow by exploiting a kernel memory corruption vulnerability (e.g., heap overflow). On the other hand, we assume that the OS kernel code integrity is guaranteed by a trustworthy hypervisor, and the hypervisor's self-protection mechanisms are enabled so that the data saved in it cannot be maliciously overwritten.

With the proposed threat model and assumptions, our goal is to eliminate (or mitigate) the possibility that an adversary can perform attacks by hijacking the control flow. Accordingly, we leverage LLVM intermediate code to construct a fine-grained CFG of kernel and limit all control-flow shifts into it. In the meantime, we save the state of the interrupted program into an interrupt stack constructed by us in the hypervisor, and validate the control data (i.e., return address) when the guest OS kernel attempts to return from an interrupt handler. Next, we describe the technical details that relate to the enforcement of CFI and the protection of control data in the interrupt context.

B. Constructing Fine-Grained CFG

To enforce fine-grained CFI for a kernel, we must obtain the fine-grained CFG of the kernel. There are three types of control flow shift in the CFG: 1) direct function call (via *direct call* instruction), 2) indirect function call (via *indirect call/jmp* instruction), and 3) function return (via *ret* instruction). Among them, the target of a direct function call and its callee's return are easy to obtain. This is because direct call instruction encodes its target address in the machine code, and the invoked function always returns to the instruction that immediately follows the call instruction. For the indirect function call, its invoked function still returns to the instruction that immediately follows the call instruction, but its target (set) cannot be directly obtained since it uses a function pointer as its target. Indirect call/jmp instruction typically needs to preload its target address into the (function) pointer, which is determined at runtime as a variable. Therefore, to obtain the fine-grained CFG, the key is to get the point-to set for every and each *indirect call/jmp* instruction.

There are two traditional approaches to get the point-to set of indirect function call. One is the static analysis of source code, and another is the runtime dynamic profiling of binary. While neither of them could solve this problem well, because the former is too complex and the latter has incomplete coverage. As the intermediate-representation (IR) code of compiler is a low-level strongly-typed language-independent representation which preserves most of the type information and enough context information that are required by our analysis, it is an effective way to determine the target set of an indirect function call by traversing the function pointer's transfer process in IR code.

In this work, we propose a retrofitted point-to analysis approach to IR, which is based on such an observation: for any *indirect call/jmp* instruction, as long as it is executed, backtracking from its function pointer, we can always find the locations where its function pointer is assigned or initialized; on the other hand, for the entry address of a function, as long as it is assigned or initialized to a function pointer, it usually (possibly through several transfers) would be consumed by one or more *indirect call/jmp* instructions. Both processes involve the transfer of function pointers, which can be leveraged to determine the point-to sets of *indirect call/jmp* instructions.

Note that for a large code base, e.g., the code base of a whole commodity operating system kernel, function signature-based policy is practical to obtain the point-to sets for indirect call/jmp instructions. In particular, the classic function signature-based policy always allows an indirect call site to invoke functions with the same function signature as itself, i.e., the same return type as well as the same number and type of arguments. Meanwhile, the classic function signature-based policy is not precise enough for a fine-grained CFI enforcement.

To improve the precision of analysis results, we introduce a new vector, called *struct location vector*, to infer the targets for each indirect call/jmp instruction. Next, we first give the formal definition of *struct location vector*, then we provide the basic worklist algorithm of our pointer analysis.

Definition 1 (Struct Location Vector): The *struct location vector* of a function pointer is the location of the function pointer member in a (nested) struct.

To better represent the cases in nested structs, we use the name of the struct, the element's order number, and the nested member's order numbers to reach the function pointer member in the struct, to denote its *struct location vector*. Thus, the *struct location vector* of a function pointer is denoted as *(%struct.name, ele_order, mem_order_*1*, …, mem_order_i, …)*, in which *ele_order* denotes the element' order number in a *%struct.name*-typed array and *mem_order_i* denotes the member's order number of layer *i* in *%struct.name* struct to reach the function pointer (i>=1). For example, vector *(%struct.sb,*0,2*)* denotes that a function pointer locates at the 2*nd* member of layer 1 in a *%struct.sb* struct, which is the 0*th* element in a *%struct.sb*-typed array.

Algorithm 1 shows the basic worklist algorithm of our pointer analysis. In the algorithm, we leverage two tables, i.e., table vector_callees.map (*VC* for short) and table signature_callees.map (*SC* for short), to record the mapping of *struct location vector/function names* and *function signature/function names* separately. According to the storing and consuming of function pointer, we proceed in two steps:

S1: For each global variable's initialization *Gi* in *IR* code, if a function pointer *fp* in *Gi* is initialized with the entry address of a function *callee*, then we put *(fp's location vector, callee)* into table *VC*, and put *(callee's function signature, callee)* into table *SC* (lines 2-5). For each instruction *I* in each function *F* in *IR* code, if the entry address of a function *callee* is assigned to a function pointer *fp* in *I*, then we put *(callee's function signature, callee)* into table *SC*; further, if *fp* is in a struct variable, then we backtrack from the assignment in *F* to get *fp*'s struct location vector, and put *(fp's location vector, callee)* into table *VC* (lines 6-12).

S2: For each indirect function callsite *ICi* in each function *F* in *IR* code, we backtrack from its function pointer *fp* until the beginning of the caller function *F* (including its arguments), and infer the point-to set of *ICi* according to three cases encountered: (1) If the function pointer *fp* is directly assigned

Algorithm 1 Basic Worklist Algorithm of Our Approach

```
 1: // Step 1: the process of function pointer storing
 2: for each Gi ∈ IR do
 3:     if fp in Gi is initialized with callee then
 4:         (fp's location vector, callee) → VC
 5:         (callee's function signature, callee) → SC
 6: for each F ∈ IR do
 7:     for each I ∈ F do
 8:         if callee is assigned to a fp in I then
 9:             (callee's function signature, callee) → SC
10:             if fp is in a struct variable then
11:                 backtrack to get fp's location vector
12:                 (fp's location vector, callee) → VC
13:
14: // Step 2: the process of function pointer consuming
15: for each F ∈ IR do
16:     for each indirect function callsite ICi ∈ F do
17:         backtrack ICi's fp to the beginning of F
18:         if fp is assigned with callee then
19:             callee → point-to-set[i]
20:         else if fp comes from a struct variable then
21:             lookup table VC, callees → point-to-set[i]
22:         else
23:             lookup table SC, callees → point-to-set[i]
```

with the entry address of a function *callee*, then we get the *callee* and add it into *ICi*'s point-to set *point-to-set[i]* (lines 18-19); (2) If the function pointer *fp* comes from a struct variable, then we look up table *VC* with *fp*'s struct location vector as input to get the callee names, and put them into *point-to-set[i]* (lines 20-21); (3) In other cases, we look up table *SC* with *fp*'s function signature as input to get the callee names, and put them into *point-to-set[i]* (lines 22-23).

The main innovation of our retrofitted point-to analysis is that we introduce *struct location vector*, to infer the targets of indirect function calls. This is reasonable as we observe that there are a large number of function pointer initializations and assignments inside struct variables in kernel space, and the function pointer locates in the same field of the struct in its lifetime, i.e., from the initialization or assignment to be consumed by indirect function calls. Combining *struct location vector* with function signature-based policy, it largely reduces the number of targets of *indirect call/jmp* instructions (average 13.14 for each *indirect call/jmp* instruction, see details in Section III-B).

In order to describe our algorithm more intuitively, we build a simple example program, as shown in Figure 1, and present the process to determine the point-to sets of indirect function calls in the example program, as shown in Figure 2.

In Figure 1(a), we show a *C* source file *example.c*. In *example.c*, it defines a global struct variable *Ga* which contains a nested struct member *b* where a function pointer *f* locates. In addition, it defines a *caller* function and four callee functions, i.e., *callee_A*, *callee_B*, *callee_C*, and *callee_D*, which will be possibly invoked by two indirect call instructions in *caller* according to the value of *flag*. In the *main* function,

```
01   typedef struct sb{
02       int i,j;
03       char* (*f)(int);} sb;
04   typedef struct sa{
05       int i;
06       sb b;} sa;
07   char *callee_A(int i) {...}
08   char *callee_B(int i) {...}
09   char *callee_C(long i) {...}
10   char *callee_D(long i) {...}
11   sa Ga={0,{0,0,callee_A}};
12   sb* Pb=&Ga.b;    int flag;
13   void caller(sb* BB, char* (*func)(long)) {
14       if(flag)
15           func=callee_C;
16       func(1);
17       BB->f(2);
18   }
19   void main() {
20       if(flag)
21           Pb->f=callee_B;
22       caller(Pb,callee_D);
23   }
```

(a)

```
01#  @Ga = global %struct.sa { i32 0, %struct.sb { i32 0, i32 0, i8* (i32)* @callee_A } }, align 8
02   define void @caller(%struct.sb* %BB, i8* (i64)* %func) #0 {
03       store %struct.sb* %BB, %struct.sb** %1, align 8
04       store i8* (i64)* %func, i8* (i64)** %2, align 8
05       ...
06#      store i8* (i64)* @callee_C, i8* (i64)** %2, align 8
07       ...
08       %7 = load i8* (i64)*, i8* (i64)** %2, align 8
09*      %8 = call i8* %7(i64 1)
10       ...
11       %9 = load %struct.sb*, %struct.sb** %1, align 8
12       %10 = getelementptr inbounds %struct.sb, %struct.sb* %9, i32 0, i32 2
13       %11 = load i8* (i32)*, i8* (i32)** %10, align 8
14*      %12 = call i8* %11(i32 2)
15   }
16   define void @main() #0 {
17       %4 = load %struct.sb*, %struct.sb** @Pb, align 8
18       %5 = getelementptr inbounds %struct.sb, %struct.sb* %4, i32 0, i32 2
19#      store i8* (i32)* @callee_B, i8* (i32)** %5, align 8
20       ...
21       %7 = load %struct.sb*, %struct.sb** @Pb, align 8
22#      call void @caller(%struct.sb* %7, i8* (i64)* @callee_D)
23   }
```

(b)

Fig. 1. An example: storing and consuming function pointers. (a) example.c; (b) LLVM IR.

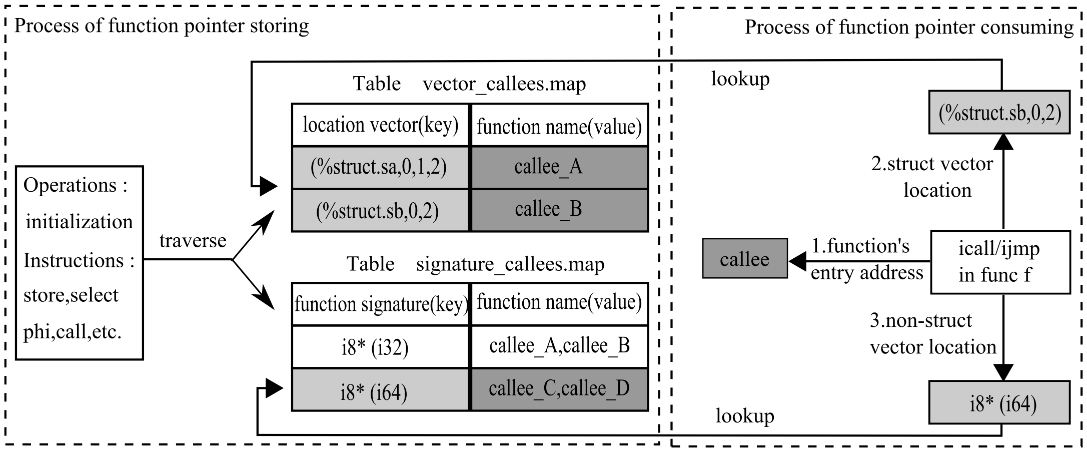


Fig. 2. The process to determine the point-to sets of indirect function calls.

it passes a struct pointer (*Pb*) and a function pointer (*callee_D*) as function arguments to *caller*. In Figure 1(b), we show the corresponding LLVM IR code of *example.c* (we omit some irrelevant content due to space limitation, such as the temporary variables %1 and %2 which are from stack allocation). The lines marked with # indicate the places where function pointer storing happens, while the lines marked with * indicate the places where function pointer consuming happens.

1) The Process of Function Pointer Storing: The process of function pointer storing needs to deal with two cases, i.e., initialization and assignment.

(1) Initialization: In Figure 1(b), line 01 defines a global variable *Ga* in which a function pointer is initialized with the entry address of function *callee_A* in a nested struct. The function pointer's location vector is *(%struct.sa,*0,1,2*)*, which denotes the 2*nd* member (*f*) inside a *%struct.sb*-typed variable (*b*) that is the 1*st* member inside a *%struct.sa*-typed variable (*Ga*) that is the 0*th* element in a struct array. As a result, we put location vector *(%struct.sa,*0,1,2*)* and function name *callee_A* into table *vector_callees.map*, as shown in Figure 2. In addition to the nested struct, a lot of aggregate data structures also need to be handled in initialization, e.g., array and vector.

(2) Assignment: In Figure 1(b), it shows three different cases where function pointers are assigned in lines 06, 19, and 22, respectively. In lines 06 and 22, the entry addresses of function *callee_C* and *callee_D* are directly assigned to a function pointer variable separately, while the entry address of function *callee_B* is assigned to a function pointer inside a struct pointer variable (lines 17-19). For the last case, we backtrack from the assignment point (line 19) until the beginning of the *main* function to get the function pointer storing location

vector *(%struct.sb,*0,2*)*. To get the location vector, we traverse a lot of fetch-related instructions including *getelementptr*, *load*, *select*, *phi*, etc. Note that *getelementptr* instruction is used to get the address of a subelement of an aggregate data structure [31]. Finally, we put location vector *(%struct.sb,*0,2*)* and function name *callee_B* into table *vector_callees.map*, as shown in Figure 2.

For all functions whose entry addresses are initialized or assigned to function pointers, we record the function signatures of them and add the pair of *function signature/function names* to table *signature_callees.map*, as shown in Figure 2. It is important to point out that only the functions in table *signature_callees.map* could possibly be indirectly invoked by indirect call/jmp instructions, but not all the functions in kernel space. After that, we have built table *vector_callees.map* and *signature_callees.map*. For table *vector_callees.map*, we add the limitation of struct type and its storing location for function pointer, thus greatly reduce the scope of possible callee set. For table *signature_callees.map*, our approach is more accurate than others which blindly match with function signature of all functions in the program, since we only allow the functions in table *signature_callees.map* to be included. Note that our approach makes a further step than FECFI [15], since we use the complete function signature (including the function's return type and the arguments list with their types) rather than only the number of function arguments to classify functions.

2) The Process of Function Pointer Consuming: In the process of function pointer consuming, we backtrack from each indirect call/jmp instruction until the beginning of caller function, and determine the source of its function pointer. In Figure 1(b), it shows two indirect call instructions in *caller* (lines 09 and 14) where function pointers are consumed. According to the source of function pointer, there are three cases as follows.

(1) Backtracking from the indirect call instruction in line 09, the function pointer %7 could be directly assigned with the entry address of function *callee_C* in line 06. So, we put *callee_C* into the point-to set of the indirect call instruction in line 09.

(2) Backtracking from the indirect call instruction in line 14, the function pointer %11 comes from a struct variable *%BB* which is an externally imported argument of *caller*. We traverse all fetch-related instructions (e.g., *getelemetptr*, *load*) encountered along the path (lines 14->13->12->11->03->02) to obtain the function pointer's location vector *(%struct.sb,*0,2*)*. Then we use this vector as input to look up table *vector_callees.map* and get the result as shown in Figure 2. In fact, the vector (%struct.sb,0,2) and (%struct.sa,0,1,2) are equivalent since they point to the same location. As a result, we put *callee_A* and *callee_B* into the point-to set of the indirect call instruction in line 14.

(3) Similar to case (1), backtracking from the indirect call instruction in line 09, the function pointer %7 could also come from an externally imported function argument *%func* (along the path of lines 09->08->04->02) rather than a struct variable. Different with the case that a function pointer is passed as an argument in a direct function call as line 22 shows, in an indirect function call, it is very difficult to figure out the function pointer which is passed as an argument. We solve this problem by inferring the targets of indirect call instruction in line 09 with the type (i.e., function signature) of function pointer %7, which is a function signature-based policy. The type of function pointer %7 is *"i8* (i64)*"* (as shown in line 08), so the callee's function signature should be *"i8* (i64)"*. We use the function signature *"i8* (i64)"* as input to look up table *signature_callees.map* and get the result *{callee_C, callee_D}* (as shown in Figure 2). Then we put them into the corresponding indirect call's point-to set. It's important to point out that in some cases, a function pointer's type might be casted to another, we have to deal with such cases carefully (the details can be found in Section III-A).

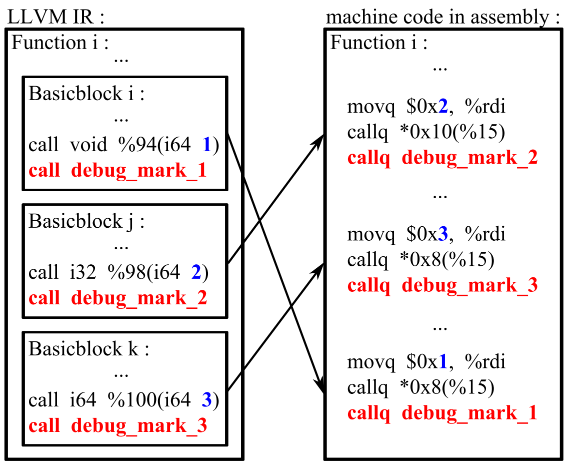


Fig. 3. Marking the indirect call instructions in a function.

After the processes of function pointer storing and function pointer consuming, we obtain the point-to sets for the indirect call instructions in Figure 1(b). The point-to set for the indirect call instruction in line 09 is *{callee_C, callee_D}*, and the point-to set for the indirect call instruction in line 14 is *{callee_A, callee_B}*. Note that our proposed point-to analysis is context-sensitive and field-sensitive, as we distinguish between different invocations of a function at different call sites, as well as different fields of a struct in *C* program.

3) Marking Indirect Call Instructions in a Function: So far we have obtained the point-to set for every indirect call/jmp instruction, however, these sets are from the IR code, but not the final machine code. We observe that there may exist more than one indirect call instruction in a function. From IR code to final machine code, it goes through a few optimizations which could change the order of the indirect call instructions in a function, or even remove one or more of them. This might alter the final CFG. Fortunately, we observe that most optimizations are based on the basic block in the function. To solve this problem, we mark each indirect call instruction with a tag right after it in the basicblock, as shown in Figure 3. First, we define some marking functions in kernel source code, like debug_mark_1, debug_mark_2, etc (the number is accumulative). Then we traverse the instructions in each basicblock in the function. When an indirect call instruction is found, we insert a tag right after it. For instance, in Figure 3, there are three indirect call instructions in *basicblock i, j, k* of *function i*, but the order of the three indirect call instructions has been changed in the final machine code. To establish the mapping of indirect call instruction in IR code and in machine code, we insert a call instruction to an ordered marking function as a tag right after each indirect call instruction in the basicblock, e.g., *call debug_mark_1* right after *call void %94(i64 1)* in *basicblock i*. By doing so, we establish the mapping of indirect call instruction in IR code and in machine code, as shown by arrows in Figure 3.

C. CFI Protection

After obtaining the fine-grained CFG, it is an intuitive thing to enforce fine-grained CFI since there have been some fairly mature solutions. We employ *indexed hooks*, which has been implemented in our previous work [28] to enforce CFI. Specifically, we create one jump table (i.e., function table) for every indirect call/jmp instruction and one jump table (i.e., return table) for each function. Every function table contains the entry addresses of the functions that may be indirectly invoked by a certain indirect call/jmp instruction, and each return table contains the legal return addresses of a certain function according to the CFG. Note that we do not need more than one return table for one function as all the return instructions in a function have the same targets. Then we instrument all the related instructions in kernel to limit every indirect control-flow shift (i.e., indirect function call or function return) into its corresponding jump table. In fact, there is an alternative approach to enforce CFI, i.e., the label-based approach proposed in the original CFI [12], and lots of CFI systems adopt label-based approach, e.g., CPI [20], KCoFI [27], Bin-CFI [13], CCFIR [14], etc. We choose *indexed hooks* rather than the lable-based approach as *indexed hooks* solved the destination equivalence problem [12], which could possibly lead to coarse-grained CFI. For technical details, refer to our previous work [28].

D. Protection of Control Data in the Interrupt Context

When an interrupt occurs, the operating system saves the interrupted program's context information on the kernel stack. Later, when the program is resumed, the context information will be restored. In the context information, it contains the CS/IP pair, which is essentially the return address, so that the program can return to the right place where it is interrupted. Note that the CS/IP pair is vulnerable to attackers since it is saved in memory. Attackers could tamper the CS/IP pair to deviate kernel's control flow, so we must provide protection to it.

However, the static analysis approach is not applicable for the protection of control data (i.e., return address) in the interrupt context, as we cannot predetermine when and where an interrupt may occur. Interrupts could occur at any time or any valid instruction boundary in both user and kernel space. To address this problem, we propose a hypervisor-based approach. We run the kernel as a guest Virtual Machine (VM) on the hypervisor, and backup the return address in the hypervisor when an interrupt arrives. Later, when the interrupt returns, we verify the return address by comparing the CS/IP pair in current kernel stack with the one saved in the hypervisor.

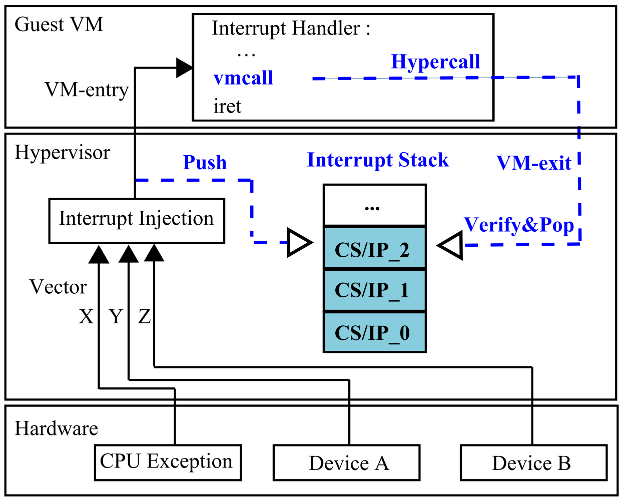


Fig. 4. The procedure of interrupt handling with our protection.

Our method is based on such an observation: all interrupts to the guest VM will be taken over by the underlying hypervisor, and the hypervisor will construct the necessary virtual interrupts and forward them to the VM. The hypervisor needs to simulate the complete process as CPU does when an interrupt arrives. For example, the hypervisor needs to push the state of the interrupted program into the current VM's kernel stack, then load the entry address of the corresponding interrupt handler into the IP register in the VM kernel, and finally the execution is transferred to the interrupt handler. Note that the hypervisor completes the above simulation with a Virtual Machine Control Structure (VMCS) which contains the interrupted program's state to be loaded into the VM.

Figure 4 shows the complete procedure of interrupt handling. The original procedure is marked with solid lines while the dotted lines are added by our method for protecting the CS/IP pair (i.e., the return address) in the interrupt context. When an interrupt arrives from an external hardware device (e.g., PIC) or internal CPU (e.g., exception), the hypervisor will take over the interrupt (if the CPU is running in VM's context at that time, it will perform *VM-exit* to return to the hypervisor), and perform interrupt virtualization and inject interrupt into the VM according to the interrupt vector. At last, before the interrupt handler returns, the CPU executes *iret* instruction to resume the program's execution. To provide protection for control data in the interrupt context, we build an interrupt stack in the hypervisor to save the CS/IP pair, and push it (which can be obtained from VMCS) into the stack when an interrupt injection happens. On the other hand, in VM OS's kernel, we instrument the *iret* instruction to verify the return address for the interrupt return. Specifically, before *iret*, we dump the return address in current kernel stack and execute a *vmcall* instruction which leads to *VM-exit* and the execution of hypercall handler in the hypervisor. Therefore, we can verify the return address by comparing the dumped value with the one on the top of the interrupt stack in the hypervisor. If it is legal, we simply pop the value in the interrupt stack. Otherwise, we transfer the control flow to a pre-defined error handler to prompt the user and block the attack. Due to the case of nested interrupt, there could be multiple CS/IP pairs in the interrupt stack simultaneously, e.g., CS/IP_0, CS/IP_1, CS/IP_2, etc, as shown in Figure 4. The instrumentation details of *iret* instruction can be found in Section III-C.

III. Implementation

To validate our approach, we have developed a proof-of-concept prototype of Fine-CFI. In this section, we discuss the implementation details as well as some additional notes we observed when developing the prototype.

Our prototype is built on top of the open-source LLVM compiler framework [30]. And we have used the prototype to enforce fine-grained CFI as well as the protection of control data in the interrupt context for the Linux 3.14/x86-amd64 kernel of LLVMLinux project [32]. In order to obtain the kernel's fine-grained CFG, we have added two passes in LLVM compiler. One is used to obtain the targets of each indirect call/jmp instruction in the kernel, and the other is for marking the indirect call/jmp instructions in a function. To protect control data in the interrupt context, we run the Linux as a guest VM in KVM hypervisor [33] (version 3.13.0). We have built an interrupt stack and redefined the hypercall's handler in KVM. Note that in our prototype, KVM is assumed to be trusted to provide kernel code integrity, as well as the protection for jump tables and the interrupt stack in itself.

A. Constructing Fine-Grained CFG

To construct fine-grained CFG, the most important thing is to obtain the point-to set for each indirect call/jmp instruction. To achieve that, we have implemented two new passes in the LLVM compiler, i.e., *point-to analysis pass* and *indirect call marking pass*.

1) Point-to Analysis Pass: Before the point-to analysis, we need to compile Linux kernel's source code to LLVM IR code and link all files together to a *vmlinux.bc* file with the linker *llvm-link*. Our point-to analysis pass operates on the LLVM IR which preserves most of the type information that is required for analysis. The analysis pass starts at a *runOnModule()* function which is launched by the LLVM optimizer *opt* and consists of two processes. In the process of function pointer storing, the pass traverses all the operations of global variables' initialization that refers to a function, as well as all the instructions (including *store, select, phi, call*, etc) which could assign the entry address of function to variables and all the transfers among those variables inside each function. During the traversing, the pass establishes two tables, i.e., table *vector_callees.map* and table *signature_callees.map*, to record the mapping of *location vector/function names* and *function signature/function names* separately. In the process of function pointer consuming, the pass backtracks from the function pointer variable of each indirect call/jmp instruction until the beginning of the function, and determines the point-to set of the indirect call/jmp instruction directly or by looking up table *vector_callees.map* or table *signature_callees.map*.

There are a few corner cases in our point-to analysis. In one case, LLVM does not preserve the original type of pointer that has been casted to *i8** when passing the pointer as a function argument. An example for such implicit cast of pointer type is shown as the following. In function *do_read_cache_page()*, an indirect call instruction invokes a function with function pointer *%filter* and the type of its first argument *%data* is *i8**. But actually *%data*'s type was casted from an original type *%struct.file** to *i8** in advance when being passed as a function argument to invoke *do_read_cache_page()*, so *%filter*'s real type is *i32 (%struct.file*, %struct.page*)**. We have augmented the compiler to preserve such type information as LLVM metadata.

- *%48 = call i32 %filler(i8* %data, %struct.page* %36)*

On the other hand, LLVM also leverages *bitcast* instruction to convert a pointer's type to another explicitly. An example for explicit cast of pointer type in function *blk_delay_work()* is shown as the following. In the example, the type of function pointer *%18* has been casted from *void (%struct.work_struct*)** to *void (%struct.request_queue*)** with a *bitcast* instruction. To address this problem, we have traversed all the *bitcast* instructions for function pointers and figured out the real type of them.

- *%17 = bitcast void (%struct.work_struct*)** %16 to void (%struct.request_queue*)***
- *%18 = load void (%struct.request_queue*)*, void (%struct.request_queue*)** %17*
- *call void %18(%struct.request_queue* %2)*

2) Indirect Call Marking Pass: To mark the indirect call instructions in a function, we have defined 200 marking functions in Linux kernel source file *linux/init/version.c* with ordered names like *debug_mark_i (i=1,2,3,...,200)*. Then we add the *indirect call marking pass* right before *CodeGenPrepare pass* which is the first optimization for LLVM IR. For each indirect call/jmp instruction in a function, the pass inserts a call instruction to an ordered function defined by us as a tag right after it in the corresponding basic block. Note that the pass needs to reset counter *i* when it enters a new function.

In the result, we have identified that there are $2,079$ functions which contain indirect call/jmp instructions. Among them, $1,364$ functions contain only one such instruction and 715 functions contain at least two of them. Among the 715 functions, the order of the indirect call instructions in 369 ones (51.6%) has been changed in the procedure from IR to the final machine code.

B. CFI Protection

After constructing the fine-grained CFG, we leverage *indexed hooks* [28] to enforce fine-grained CFI with this CFG. Specifically, based on the observation that existing control data (e.g., function pointers or return addresses) have a set of legitimate jump targets, *indexed hooks* precalculates them into (protected) jump tables and then replaces these control data with their indexes to these tables. Note that the jump tables are read-only and thus can be protected with the hardware-based page-level protection. To do that, we have developed a new class in LLVM backend to identify and instrument all the control data-related instructions (e.g., direct call, indirect call/jmp, and ret instructions), as well as a tool to identify and

```
ffffffff8141329a <irq_return>:
ffffffff8141329a: 48 cf                 iretq
              is instrumented as
ffffffff8141329a <irq_return>:
ffffffff8141329a: 41 54                 push  %r12
ffffffff8141329c: 41 55                 push  %r13
ffffffff8141329e: 4c 8b 64 24 10        mov   0x10(%rsp), %r12   #RIP
ffffffff814132a3: 4c 8b 64 24 18        mov   0x18(%rsp), %r13   #CS
ffffffff814132a8: 0f 01 c1              vmcall
ffffffff814132ab: 41 5d                 pop   %r13
ffffffff814132ad: 41 5c                 pop   %r12
ffffffff814132af: 48 cf                 iretq
```

Fig. 5. An instrumentation example for interrupts.

replace the entry addresses of callees stored in both data and code sections in the kernel image.

In the result, we have replaced 50,677 direct call, 4,074 indirect call/jmp, and 14,488 ret instructions in the kernel. Accordingly, we have created 4,074 function tables and each of them is for one indirect call/jmp instruction. Among these function tables, 447 of them (11.0%) have only one target. The indirect call with the most targets (315 targets) is in the *psmouse_attr_set_helper()* function and it is responsible for invoking different helpers through *attr->set* function pointer to set different attributes. *The average number of targets for one indirect call/jmp instruction is only 13.14!* Also, we have created 11,906 return tables and each of them is for one function. Among the return tables, 3,424 of them (28.8%) have only one return target, and the average number of return targets for one function is 8.76.

In order to apply hardware-based page-level protection, all the jump tables constructed from the fine-grained CFG need to be put together in page-aligned memory and marked as read-only. All unoccupied entries in the jump tables are filled with the entry address of *error_handler()* function, which is defined by us to trap the illegal control transfers. To store the jump tables, we need 65.5MB extra memory space.

C. Protection of Control Data in the Interrupt Context

In order to protect control data in the interrupt context, we need to make changes both in the hypervisor and in the VM. In the hypervisor, we have extended KVM to backup and verify the CS/IP pair saved in the interrupt context. While in the VM, we have modified the VM kernel to dump the CS/IP pair in current kernel stack before the kernel returns from an interrupt, then exit from the VM and enter the hypervisor. Specifically, in the source file *arch/x86/kvm/vmx.c* of KVM, we construct an interrupt stack and define a *vmx_push_CS_IP()* function to push the CS/IP pair for interrupt into the interrupt stack when we detect an interrupt injection. Then we redefine KVM's hypercall handler to verify the CS/IP pair and return to the VM. In KVM, *kvm_emulate_hypercall()* function is used to handle the hypercall request which could be launched by a *vmcall* instruction in the guest VM kernel.

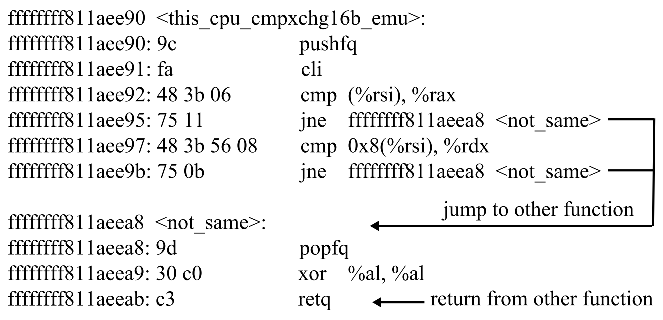
In the source file *arch/x86/kernel/entry_64.S* of Linux VM, we insert seven instructions before *iret* instruction in function *irq_return()*, as shown in Figure 5. We use two *mov* instructions (*mov 0x10(%rsp),%r12* and *mov 0x18(%rsp),%r13*) to dump the CS/IP pair in current VM kernel stack (in *0x10(%rsp)* and *0x18(%rsp)*) into *%r12* and *%r13* registers, which are pre-pushed into the stack (with *push %r12* and *push %r13*) for later restoring (with *pop %r13* and *pop %r12*). After that, a *vmcall* instruction is executed to exit from the VM and enter the hypervisor, so that we can verify the dumped data in *%r12* and *%r13* registers. Note that there could exist a small window of race condition from the other processors if the VM is running on an SMP machine. However, this window is very small and largely unpredictable.

D. Additional Prototyping Notes

1) Loadable Kernel Module Support: In our prototype, we only allow the trusted loadable kernel modules (LKMs) to be loaded into base kernel. We could accomplish it by simply integrating the existing module loading schemes, e.g., NICKLE [34] or SecVisor [35]. The key challenge is about unifying fine-grained jump tables in kernel and loaded/unloaded LKMs. To achieve that, first we need to perform point-to analysis for the loaded module to obtain its fine-grained CFG and enforce fine-grained CFI with this CFG in the module. When the module is loaded, a fix-up routine runs to merge the module's CFG into kernel's, then computes the new jump tables from the merged CFG. Note that we need to fix up the base addresses of jump tables in kernel code base since some jump tables' size has been changed after merging. On the other hand, when a module is unloaded, the fix-up routine runs to replace its indexed entries in the new jump tables temporarily with the address of error handler to trap abandoned control transfers. Note that the fix-up routine is protected by the hypervisor to make it secure.

2) Point-to Analysis for Indirect Calls in Assembly Code: The Linux kernel source tree consists of most of C files and a few assembly files (e.g., *arch/x86/kernel/entry_64.S*). We have found that there are 7 indirect call instructions in assembly source code in our prototype. Since the point-to analysis operates on IR code, in which assembly code has not yet been included, we have to determine the point-to sets for these 7 indirect call instructions with another approach. Fortunately, the amount of such instructions is small and we can deal with them manually. Interestingly, for the indirect call instruction in *system_call_fastpath()* function in *arch/x86/kernel/entry_64.S* file, we found that all the target functions begin with a prefix *SYS_* or *stub_* (e.g., *SYS_read()* or *stub_fork()*), each of which represents the target of a system call.

3) Context Switch: In Linux, the scheduler needs to construct context information for the next scheduled process that was interrupted before and restart it with an *iret* instruction, which leads to a context switch. In such case, the order of the CS/IP pairs saved in the interrupt stack constructed by us in KVM may be altered. Note that our verification of return address in the interrupt context always uses the CS/IP pair on the top of the stack for comparison, which may lead to a false. To solve this problem, we match every entry from top to bottom in the interrupt stack. If the match succeeds, we copy the top entry to the current location and then pop the top entry.



```
ffffffff811aee90 <this_cpu_cmpxchg16b_emu>:
ffffffff811aee90: 9c              pushfq
ffffffff811aee91: fa              cli
ffffffff811aee92: 48 3b 06        cmp  (%rsi), %rax
ffffffff811aee95: 75 11           jne  ffffffff811aeea8 <not_same>
ffffffff811aee97: 48 3b 56 08     cmp  0x8(%rsi), %rdx
ffffffff811aee9b: 75 0b           jne  ffffffff811aeea8 <not_same>

ffffffff811aeea8 <not_same>:
ffffffff811aeea8: 9d              popfq
ffffffff811aeea9: 30 c0           xor  %al, %al
ffffffff811aeeab: c3              retq
```

Fig. 6. An example of broken convention of function call.

4) Signal Handler Dispatch: The operating system often needs to dispatch signal handlers for a process. To do that, before returning from an interrupt, the kernel invokes *do_signal()* function to construct context information for the process including filling the CS/IP fields in current kernel stack with the corresponding CS/signal handler's entry address. To protect the above control data in the signal handler dispatch, we do the same instrumentation in *do_signal()* function as that in Figure 5 to dump the control data from the current kernel stack, and then backup it into the interrupt stack in KVM for later verification.

5) Broken Convention of Function Call: In our prototype, we encountered a special case which needs to be further handled. Conventionally, if a function (e.g., A) is invoked somewhere, then it should return from the same function (i.e., A) by executing a *ret* instruction. But the convention could be broken in some special cases, namely it may return from a different function rather than A. As shown in Figure 6, when *this_cpu_cmpxchg16b_emu()* function is invoked in one place, it could return from *not_same()* function rather than itself by executing a *retq* instruction at $0xffffffff811aeeab$. To address this case, we instrument the *retq* instruction in *not_same()* function instead of *this_cpu_cmpxchg16b_emu()* to limit the control transfers. Fortunately, such cases only happen four times and we fix them manually.

IV. Evaluation

To obtain the fine-grained CFG for Linux kernel, our prototype has added $3,027$ lines of C++ code to LLVM. There is also a need to modify Linux kernel source code and KVM to protect control data in the interrupt context. Our prototype has added 12 lines of assembly code to Linux source code and 134 lines of C code to KVM separately. Compared to the original kernel image, our prototype image file size increased from $53,710,617$ bytes to $56,442,438$ bytes (5.1%) and the number of instructions increased from $956,008$ to $1,121,848$ (17.3%).

Next, we perform security evaluation to our system and present the performance measurement results.

A. Security Evaluation

We have conducted two empirical evaluations and two attack vectors to measure the security of our system.

The first empirical evaluation shows how well that Fine-CFI removes instructions from the set of instructions that could be leveraged in a return-oriented programming attack (which can be performed with or without return instructions [4], [7], [8]). We compute a metric, i.e., average indirect target reduction (AIR) [13], to show this reduction. The second empirical evaluation shows the number of remaining gadgets after our instrumentation and if they could be used in an attack. We leverage an open source tool (i.e., ROPgadget [36]) for the purpose.

Additionally, we perform two attack vectors against Fine-CFI. In the first attack, we use RIPE benchmark [37] to evaluate the defense of Fine-CFI against control flow hijacking. In the second attack, we tamper the control data in the interrupt context at certain time which could be corrupted with a memory error in Linux VM kernel for evaluation.

1) Average Indirect Target Reduction: ROP attacks work because they can misuse too many available instructions which can be executed within a program. What is worse, in certain hardware architecture such as x86, because of the variable-length encoding and unaligned execution of machine instructions, every byte in the code segments could be a possible target of indirect control transfer. To better understand how well Fine-CFI removes instructions from the set of instructions that could be used in a ROP attack, we employ Zhang and Sekar's AIR metric [13].

Equation 1 from Zhang and Sekar [13] shows the general form to compute the AIR metric for a program. In the equation, n denotes the number of indirect control transfer instructions (i.e., indirect call/jmp and ret) in the program, S denotes the total number of possible targets to which an indirect control transfer instruction can jump in an unprotected program, and $|T_j|$ denotes the number of targets to which indirect control transfer instruction j can jump after protection:

$$\frac{1}{n}\sum_{j=1}^{n}(1-\frac{|T_j|}{S}) \tag{1}$$

Since every indirect control transfer instruction j instrumented by Fine-CFI can transfer control flow to its own set of addresses (with number $|T_j|$), Equation 1 can be simplified into Equation 2 (with |T| being the total number of legal targets for all indirect control transfer instructions which is equal to $\sum_{j=1}^{n}|T_j|$):

$$1-\frac{|T|}{n*S} \tag{2}$$

We have computed the AIR metric for the code within the Linux kernel image that generated by Fine-CFI. In particular, Fine-CFI has identified $1,057,863$ legal targets ($|T|$) for all indirect control transfers out of $3,590,398$ possible targets ($|S|$) in the kernel's code segments before instrumentation. The image generated by LLVM contains $18,562$ indirect control transfer instructions (n) which include indirect call/jmp and ret instructions. The average reduction of targets (AIR metric) for these transfers is therefore 99.998%. We believe that nearly all the possible targets of indirect control transfer have been eliminated as feasible targets by Fine-CFI.

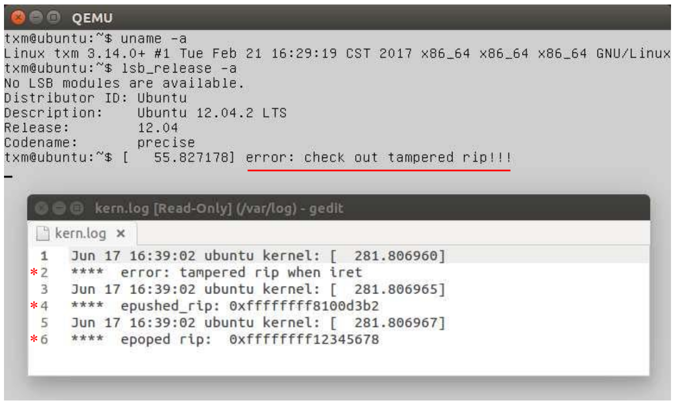
We have made a comparison of the reduction of the target branch set to three previous works in this area. The results show the AIR metric of Fine-CFI (99.998%) is better than bin-CFI [13] (98.86%), KCoFI [27] (98.18%) and Ge et al.'s work [38] (99.6% for return targets and 99.1% for indirect call/jmp targets).

2) ROP Gadgets: Note that it is possible for an attacker to perform code-reuse attacks with only a handful of gadgets within the remaining potential targets. What is important is to figure out the number of the remaining gadgets and how useful they are in an attack. To achieve that, we used the open-source tool ROPgadget [36] (version 5.6) to automatically analyze the ROP gadgets remaining in both the original and instrumented Linux kernel. We ran the tool using the *--range* command line option to make sure that all the found gadgets are included in code sections. In the result, ROPgadget found 44, 063 gadgets in the original Linux kernel and 841 gadgets which start with aligned instruction addresses in the instrumented Linux kernel. We checked all the 841 gadgets and confirmed that none of them can be reached through a control flow which is allowed by our fine-grained CFG. Therefore, none of these gadgets can be used to launch an effective ROP attack.

3) Control-Flow Hijack Attacks: To evaluate the effectiveness of FINE-CFI for defending against control-flow hijacks, we used RIPE test suite [37]. RIPE is a benchmark containing a dumb program with vulnerabilities which can be exploited by 850 attack forms to hijack the program's control flow. We have performed attacks with RIPE benchmark on several systems. Specifically, on a Ubuntu 6.06 system without any built-in defense mechanisms, almost all 850 attacks can succeed (a few ones succeed with a certain probability). On a Ubuntu 14.04 (Trusty Tahr) system, with all the built-in defense mechanisms disabled (including DEP, ASLR, and stack cookies), 288 attacks succeed. With all defense mechanisms enabled, 54 attacks succeed. While on the Ubuntu 14.04 (Trusty Tahr) system with all the built-in defense mechanisms disabled but our fine-grained CFI enabled, none of attacks succeed. As a result, FINE-CFI prevents all the attacks that hijack the control flow in the program of RIPE.

4) Attack on Control Data in the Interrupt Context: We have simulated an attack on control data in the interrupt context. Specifically, after having pushed return address of interrupt into the interrupt stack in KVM 60, 000 times by *vmx_push_CS_IP()*, we tampered the RIP in VMCS with a self-defined value $0xffffffff12345678$. After *VM-entry*, the modified RIP would be pushed into the current kernel stack of the VM. As a result, before returning from the interrupt, our system has detected an attack when performing verification and transferred the VM's control flow to *error_handler()* function. As shown in Figure 7, in the VM, *error_handler()* function printed the error information (i.e., *"error: check out tampered rip!!!"* which has been underlined) to prompt user, and blocked the attack. The detailed log is available in */var/log/kern.log* file in the hypervisor (lines 1-6). In particular, in *kern.log*, we found that an error occurred in line 2 (marked with *). The original pushed return address (i.e., RIP) is $0xffffffff8100d3b2$ (line 4 marked with *), but it has been modified by the attacker to another value, i.e., $0xffffffff12345678$ (line 6 marked with *).

It is worth mentioning that it has been shown that SMEP/SMAP [39], [40] could be bypassed with kernel



```
txm@ubuntu:~$ uname -a
Linux txm 3.14.0+ #1 Tue Feb 21 16:29:19 CST 2017 x86_64 x86_64 x86_64 GNU/Linux
txm@ubuntu:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 12.04.2 LTS
Release:        12.04
Codename:       precise
txm@ubuntu:~$ [   55.827178] error: check out tampered rip!!!
_
```

```
 1   Jun 17 16:39:02 ubuntu kernel: [  281.806960]
*2   ****  error: tampered rip when iret
 3   Jun 17 16:39:02 ubuntu kernel: [  281.806965]
*4   ****  epushed_rip: 0xffffffff8100d3b2
 5   Jun 17 16:39:02 ubuntu kernel: [  281.806967]
*6   ****  epoped rip:  0xffffffff12345678
```

Fig. 7. An attack on control data in the interrupt context.

code-reuse attacks, although they are effective hardware countermeasures to classic ret2usr attacks. Popov provides a method to bypass SMEP by tampering the function pointer with kernel function *native_write_cr4()* and passing a controlled argument with the bit 20 of CR4 register cleared to disable it [41]. Nikolenko shows another trick to disable SMEP by chaining several useful kernel gadgets and pivoting the stack for clearing the bit, or bypass SMAP by abusing vDSO [42]. However, FINE-CFI prevents both the kernel code-reuse attacks and the ret2usr attacks, by providing the fine-grained CFI protection and control data protection in the interrupt context. Thus, FINE-CFI is immune to the above attacks.

B. Performance Evaluation

To evaluate the performance overhead introduced by FINE-CFI, we have performed several benchmark-based measurements including Phoronix Test Suite [43], LMbench [44], UnixBench [45], and SPEC CPU2006 benchmark [46]. We ran our tests on a Dell Z620 workstation with an Intel Xeon CPU (12 cores @2.00GHZ) and 24GB memory. For each benchmark, we load the Linux 3.14/x86-amd64 kernel in KVM with three versions: the original version (*Original*), the new kernel with fine-grained CFI for all the indirect call/jmp and ret instructions (*New-f*), and the new kernel/KVM with fine-grained CFI for all the indirect call/jmp and ret instructions as well as the protection of control data in the interrupt context (*New-f+i*). Table I lists the configurations of the software used in our evaluation. We ran each test 10 times and calculated the average.

First, to better measure the impact on real-world applications by our system, we select five representative real-world applications in the Phoronix Test Suite, i.e., compress-7zip, postmark, nginx, cachebench, and network-loopback, which stand for the performance behaviors in processor, disk, system, memory, and network, respectively, to evaluate the performance overhead incurred by FINE-CFI. Table II lists the performance overhead results for the five real-world applications in Phoronix Test Suite. *New-f* incurs 5.39%/8.96% (average/maximum) overhead while *New-f+i* incurs 9.89%/15.36% (average/maximum) overhead. Compared to KCoFI [27], for postmark benchmark which is used to measure

TABLE I
SOFTWARE CONFIGURATIONS FOR EVALUATION

Item	Version	Configuration/Command
Phoronix Test Suite	7.4.0	phoronix benchmark ***
LMbench	3-alpha4	make results see
UnixBench	5.1.2	./Run
SPEC CPU2006	1.0.1	Runspec - -size=ref - -iterator=10 int/fp

TABLE II
REAL-WORLD APPLICATION RESULTS WITH PHORONIX TEST SUITE

Benchmark	Original	New-f	New-f+i
compress-7zip (MIPS)	2256	2174 (3.63%)	1910 (15.34%)
postmark (TPS)	338	321 (5.03%)	312 (7.69%)
nginx (Req/s)	8405.92	8194.53 (2.51%)	8110.35 (3.52%)
cachebench (MB/s)	7950.12	7406.55 (6.84%)	7349.50 (7.55%)
network-loopback (s)	34.70	37.82 (8.96%)	40.03 (15.36%)
average overhead:		(5.39%)	(9.89%)

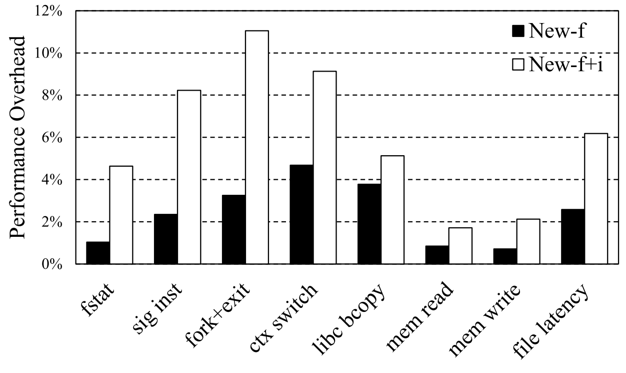


Fig. 8. Micro-benchmark results with LMbench.

the file system performance, FINE-CFI only incurs 7.69% overhead while it incurs 1.96x on average.

Second, Figure 8 shows the performance overhead of eight kernel tasks in LMbench [44]. The tasks consist of system calls, process creation, context switch, local communication, memory operations, and file latency. Among the results, the maximum performance overheads are 4.68% for *New-f* and 11.05% for *New-f+i*, which occur in the context switch and fork+exit operations, respectively. Compared to KCoFI [27], FINE-CFI has a better performance as KCoFI incurs 3.50x overhead in fork+exit operation.

Third, in UnixBench measurement, the final scores for *Original*, *New-f*, and *New-f+i* are 1, 813.8, 1, 767.7 (2.54%), and 1, 683.2 (7.20%), respectively. Compared to Ge et al.'s work [38] which also evaluates with UnixBench, FINE-CFI incurs 7.20% overhead for a Linux kernel while it incurs 11.91% overhead for a FreeBSD kernel.

Fourth, to evaluate compute-intensive performance on real applications, we use SPEC CPU2006 benchmark [46] to measure the performance overhead imposed by FINE-CFI. Table III lists the performance overhead results for SPECint. In the results, the maximum performance overheads are 12.56% for *New-f* and 18.88% for *New-f+i*, which occur in the 458.sjeng and 401.bzip2 benchmarks respectively. Table IV lists the performance overhead results for SPECfp, where the

TABLE III
SPECINT 2006 BENCHMARK PERFORMANCE BY FINE-CFI FLAVOR

Benchmark	Original	New-f	New-f+i
400.perlbench (s)	539	530 (−1.67%)	551 (2.23%)
401.bzip2 (s)	678	706 (4.13%)	806 (18.88%)
403.gcc (s)	450	464 (3.11%)	477 (6.00%)
429.mcf (s)	863	855 (−0.93%)	872 (1.04%)
445.gobmk (s)	730	708 (−3.01%)	767 (5.07%)
456.hmmer (s)	699	718 (2.72%)	732 (4.72%)
458.sjeng (s)	772	869 (12.56%)	889 (15.16%)
462.libquantum (s)	533	527 (−1.13%)	550 (3.19%)
464.h264ref (s)	822	860 (4.62%)	881 (7.18%)
471.omnetpp (s)	518	511 (−1.35%)	549 (5.98%)
473.astar (s)	704	655 (−5.54%)	762 (8.24%)
483.xalancbmk (s)	365	391 (7.12%)	408 (11.78%)
average overhead:		(1.72%)	(7.46%)

TABLE IV
SPECFP 2006 BENCHMARK PERFORMANCE BY FINE-CFI FLAVOR

Benchmark	Original	New-f	New-f+i
433.milc (s)	660	636 (−3.64%)	688 (4.24%)
444.namd (s)	621	568 (−5.64%)	676 (8.86%)
447.dealII (s)	478	486 (1.67%)	551 (15.27%)
450.soplex (s)	350	369 (5.43%)	385 (10.00%)
453.povray (s)	268	269 (0.37%)	274 (2.24%)
470.lbm (s)	572	586 (2.45%)	594 (3.85%)
482.sphinx3 (s)	885	897 (1.36%)	981 (10.85%)
average overhead:		(0.29%)	(7.90%)

TABLE V
PERFORMANCE FOR ENFORCING CFI IN SPECCPU BENCHMARKS

Benchmark	Original	New-icall	New-ret	New-f
401.bzip2 (s)	585	588 (0.51%)	625 (6.84%)	627 (7.18%)
403.gcc (s)	363	366 (0.83%)	375 (3.31%)	378 (4.13%)
429.mcf (s)	798	809 (1.38%)	880 (10.28%)	894 (12.03%)
458.sjeng (s)	721	729 (1.11%)	785 (8.88%)	794 (10.12%)
average overhead:		(0.96%)	(7.33%)	(8.37%)

450.*spolex* benchmark shows the maximum performance overhead (5.43%) for *New-f*, and the 447.*dealII* benchmark shows the maximum performance overhead (15.27%) for *New-f+i*. Interestingly, compared to the original system, in some cases the performance of *New-f* is slightly improved after deploying the fine-grained CFI. For example, in 444.*namd* benchmark, the performance is improved by 5.64% in *New-f*. We believe the possible reason for that is that the jump tables used in FINE-CFI are better localized and the cache utilization is improved.

Fifth, to further demonstrate the performance overhead introduced by FINE-CFI, we have implemented our approach to four benchmarks in SPEC CPU2006 benchmarking suite, i.e., 401.bzip2, 403.gcc, 429.mcf, and 458.sjeng for evaluation. For convenience, we ran these tests on the Dell Z620 workstation directly. These benchmarks depend on the *libc* library as well as some objects in C runtime library (e.g., crt1.o, crti.o, and crtn.o). To achieve that, we need to instrument the benchmark programs as well as the dependencies. Table V lists the performance overhead results incurred by our instrumentation. On average, *New-f* incurs 8.37% overhead. Further, 429.mcf benchmark shows the maximum overhead which is 12.03%, indicating that the vehicle scheduling for network transport has traversed the most instructions we instrumented.

Sixth, for the above C benchmark programs (i.e., 401.bzip2, 403.gcc, 429.mcf, and 458.sjeng), we have also achieved the goal to break the results down by only instrumenting indirect call/jmp (*New-icall*) or call/ret (*New-ret*) instructions, to see the impact on the forward/backward edge protection. The results (listed in Table V) show that *New-ret* incurs most of the overhead (7.33% on average) while *New-icall* incurs very little (0.96% on average), and the sum overhead (8.29%) incurred by *New-icall* and *New-ret* is roughly equal to the overhead incurred by *New-f* (8.37%), which is reasonable. We believe that the overheads introduced by *New-icall* and *New-ret* depend on the executed times of our instrumented indirect call/jmp and ret instructions.

In summary, FINE-CFI is a lightweight and fine-grained CFI system that introduces less than 10% performance overhead on average. Among them, two additional memory accesses for jump tables and two VM operations (i.e., VM-exit and VM-entry) are the main factors that cause the performance overhead.

V. DISCUSSION

In this section, we examine possible limitations of FINE-CFI and suggest potential future work.

First, to enforce fine-grained CFI for operating system kernels, we have used a compiler-based approach that requires to recompile the kernel source code to confine kernel's indirect control-flow shifts into its CFG, which needs access to the kernel source code. Moreover, for the point-to analysis to obtain fine-grained CFG, our approach also needs access to the kernel IR code compiled from its source code. However, we believe this is necessary to enable the identification of all indirect control transfers and then apply protection on them, which is not available in other dynamic profiling approaches [47], [48]. As pointed out in [28], the completeness is important since attackers can hijack kernel's control flow by deviating only *one* indirect control transfer to launch their attacks.

Second, by converting control data into indexes, FINE-CFI changes the semantics of function pointers and return addresses. Specifically, the original call instruction pushes a return address on the stack and then jumps to the destination, while our scheme pushes a table index instead and then jumps. On the other hand, in original convention every function pointer stores a function address, while our scheme replaces the address with a table index. This instrumentation has changed the semantic of the data. In some special cases, it might lead to a fault. For example, when a function pointer is compared with another value by a *cmp* instruction, instead of only being consumed by an indirect call/jmp instruction. Although so far we have not encountered such cases in our prototype, we can recover the original control data with its index before such usage.

Third, FINE-CFI cannot provide protection for dynamically generated code which is widely used in just-in-time (JIT) compilation and dynamic binary translation (DBT). However, as far as we know, nearly no CFI enforcement can work for dynamically generated code since it violates the base assumption of CFI, i.e., code integrity. The code cache used to store generated code is writable and executable either at the same time or alternately, which possibly leads to an exploitation that could be leveraged by attackers to break code integrity. As a result, to protect dynamically generated code, we have to combine other approaches (e.g., SDCG [49]) which guarantee the generated code integrity as a complementary solution to FINE-CFI.

Fourth, FINE-CFI aims to provide fine-grained CFI for operating system kernels, and non-control data attacks [50] are out of the scope in this work. In order to perform a non-control data attack, an attacker usually needs to hijack the program's control flow to jump to her injected malicious code or misuse the existing code snippets. FINE-CFI can prevent both cases of them. Note that researchers have proposed a few solutions to defeat non-control data attacks for kernel extensions [51] or provide protection for a subset of non-control data in kernels [52], [53]. Nevertheless, how to provide comprehensive and efficient protection to non-control data for operating system kernels is still an open question.

VI. RELATED WORK

Since the introduction of original CFI by Abadi et al. [12] in 2005, a variety of CFI systems have been proposed to address control-flow hijacking attacks. Among these systems, most of them are implemented in user level (or for a tiny hypervisor), e.g., Control Flow Locking [54], CCFIR [14], Bin-CFI [13], Strato [55], MIP [56], FECFI [15], CPI [20], O-CFI [21], PICFI [57], Context-sensitive CFI [58], CCFI [22], HyperSafe [26], etc, thus cannot be readily deployed in kernels due to additional unique challenges and complexities in the OS kernel. Moreover, almost all the user-level (and hypervisor-level) CFI invariants do not provide protection for the control data in kernel space, thus are vulnerable to ret2usr [25] attacks. In contrast, FINE-CFI aims to enforce fine-grained CFI for operating system kernels, and it also provides protection for control data in the interrupt context, thus defeats the ret2usr and kernel code-reuse attacks.

On the other hand, researchers have also designed mechanisms to protect control data or enforce CFI in kernel space. For example, Lares [59] and HookSafe [47] protect a subset of function pointers in kernel from being hijacked. Returnless [60] instead protects the return addresses from being misused to prevent kernel-level return-oriented rootkits. These systems cannot enforce complete CFI as they only provide protection for a subset of kernel control data.

To enforce CFI for the whole OS kernel, SBCFI [61] leverages a hypervisor to dynamically detect the violation of kernel control-flow integrity, e.g., by comparing with a statically computed CFG. However, by design, SBCFI detects the violation *after* the system is compromised.

SVA [62] proposes an efficient and robust approach to provide a safe execution environment for both kernel and application code. SVA provides stronger security guarantees than FINE-CFI as it enforces strong memory safety properties. However, as pointed out in [27], the performance overhead introduced by SVA is relatively high even after being optimized by techniques using sophisticated whole-program pointer analysis, which impedes its practical deployment. In contrast, FINE-CFI enforces fine-grained CFI as well as the protection of control data in the interrupt context with less than 10% average performance overhead.

KCoFI [27] is a system building upon SVA [62], which enforces complete CFI for commodity operating system kernels without using heavyweight complete memory safety, thus, it introduces far lower overhead than SVA. Specifically, KCoFI uses the SVA compiler instrumentation capabilities and the SVA-OS instruction set to identify and control both OS kernel/hardware interactions and OS kernel/application interactions. However, KCoFI only provides coarse-grained CFI protection as it uses only one label for the targets of all the indirect call/jmp and return sites. In contrast, FINE-CFI leverages a retrofitted point-to analysis to obtain the fine-grained CFG of the kernel and establishes one jump table for each indirect control transfer instruction according to this CFG. When an indirect control transfer occurs, its targets will be confined to its corresponding jump table. By doing so, FINE-CFI enforces a much stricter CFI policy than KCoFI. In particular, FINE-CFI reduces the number of indirect control-flow targets by 99.998%, which is much better than KCoFI (98.18%).

Indexed hooks [28] provides comprehensive and efficient protection for kernel control data, which essentially enforces CFI for operating system kernels. In particular, to obtain the fine-grained CFG, *indexed hooks* determines the point-to set of each indirect call/jmp instruction using a dynamic profiling approach. As mentioned earlier, as the dynamic analysis has an incomplete coverage, *indexed hooks* conservatively assumes a maximum set for those unreachable indirect function calls, which could possibly lead to a coarse-grained CFI. In contrast, the main contribution of FINE-CFI is that it leverages a retrofitted context-sensitive and field-sensitive pointer analysis (rather than the dynamic analysis in *indexed hooks*) which greatly improves the precision of kernel CFG. Note that to enforce fine-grained CFI for an operating system kernel, the key is to get a precise and fine-grained CFG of it.

Recently, a closely related work [38] adopts a similar scheme (restricted pointer indexing) to FINE-CFI for enforcing kernel CFI protection. In particular, it leverages the conservative function pointer usage patterns found in the kernel source code to develop a method to compute fine-grained CFGs for kernels. In contrast, the biggest advantage of FINE-CFI is that our pointer analysis is based on LLVM IR code while that work is based on source code. As mentioned in Section II-B, it is effective to determine the target set of an indirect function call by traversing the function pointer's transfer process in IR code due to its rich type and context information. As a result, we get a higher precision. For instance, the AIR metric of FINE-CFI is 99.998%, which is better than that work (99.6% for return targets and 99.1% for indirect call/jmp targets).

RAP [63] is a commercial product to secure Linux kernel by enforcing fine-grained CFI. Compared to RAP, FINE-CFI makes improvement in three aspects. First, RAP generates hashes for functions based on their prototype, regardless of the fact that a respective function pointer may not exist. In contrast, FINE-CFI only allows the functions in table *signature_callees.map* (whose addresses have been taken) to be included, thus it improves the precision. Second, RAP leverages a similar label-based (i.e., hash-based) approach proposed in the original CFI [12] to enforce protection, thus it suffers the issue of "destination equivalence" which could possibly lead to coarse-grained CFI. In contrast, FINE-CFI solves the problem using the *indexed hooks* approach. Third, FINE-CFI proposes a hypervisor-based approach to protect control data in the interrupt context, thus is immune to ret2usr attacks while RAP does nothing at this point.

seL4 [64] is a formal proof of function correctness of a microkernel. Thus, seL4 provides stronger security guarantees than FINE-CFI. But it only enforces them on a microkernel comprising approximately $8,700$ lines of *C* code and 600 lines of assembly code (note that seL4 assumes the correctness of assembly code), which is much smaller than a general-purpose commodity operating system kernel protected by FINE-CFI, i.e., the Linux kernel. Moreover, changing to seL4 code is a challenge as it possibly needs manual updates to the correctness proof while FINE-CFI can address this issue automatically.

On the attack side, two attacks targeting fine-grained CFI have been proposed, i.e., Control-Flow Bending (CFB) [65] and Control Jujutsu [66]. Specifically, CFB and Control Jujutsu misuse some dispatchers (e.g., *printf()* function or the argument corruptible indirect call sites) that could change their own return addresses or function pointers to deviate program's control flow, even with the protection of fine-grained CFI. Note that both CFB and Control jujutsu are implemented in user level, their deployment in kernel space still remains to be demonstrated. Supposing CFB and Control Jujutsu could work in kernel space theoretically, FINE-CFI is also able to mitigate such attacks effectively. First, FINE-CFI limits both the targets of every indirect call/jmp instruction (i.e., the forward edge of CFG) and the return targets of each function (i.e., the backward edge of CFG). In particular, FINE-CFI creates one function table for every indirect call/jmp instruction which contains its valid targets, and one return table for each function which contains its legal return targets according to the fine-grained CFG we obtained. In our prototype, the average number of targets for one indirect call/jmp instruction is only 13.14 and the average number of return targets for one function is only 8.76 (see Section III-B), thus it largely reduces the number of targets of dispatchers misused by attackers. Additionally, as pointed out by the authors of Control Jujutsu [66], the main reason making their attack feasible is that the Data Structure Analysis (DSA) pointer analysis algorithm [67] loses context sensitivity and field sensitivity when constructing the forward-edge of CFG, which is not the case for FINE-CFI. In summary, FINE-CFI largely raises the bar for attackers to hijack the control flow in operating system kernels.

VII. CONCLUSION

In this paper, we present FINE-CFI, a system to enforce fine-grained control-flow integrity for operating system kernels. Specifically, FINE-CFI leverages a retrofitted pointer analysis to obtain the point-to set of each indirect call/jmp instruction and further constructs the fine-grained CFG of the kernel, then it uses the CFG to enforce fine-grained CFI for the kernel. As well, FINE-CFI proposes a hypervisor-based approach to provide protection for control data in the interrupt context. With both of them, FINE-CFI effectively defeats the ret2usr and kernel code-reuse attacks. To validate our approach, we have developed a proof-of-concept prototype with LLVM compiler and enforced protection for Linux 3.14/x86-amd64 kernel. The evaluation results demonstrate the effectiveness and efficiency of FINE-CFI.

REFERENCES

[1] S. Andersen and V. Abella. (2004). *Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2 Part 3: Memory Protection Technologies*. Accessed: Jul. 6, 2017. [Online]. Available: http://technet.microsoft.com/en-us/library/bb457155.aspx

[2] A. van de Ven and I. Molnar, "Exec shield," *Retr. Mar.*, vol. 1, p. 2017, Mar. 2004.

[3] R. Wojtczuk, "The advanced return-into-lib (c) exploits: Pax case study," *Phrack Mag.*, vol. 11, pp. 4–14, 2001.

[4] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2007, pp. 552–561.

[5] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, 2008, pp. 27–38.

[6] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proc. USENIX Secur. Symp.*, 2009, pp. 1–16.

[7] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 559–572.

[8] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. 6th ACM Asia Conf. Comput. Commun. Secur.*, 2011, pp. 30–40.

[9] PaX-Team. (2003). *Pax ASLR (Address Space Layout Randomization)*. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[10] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2013, pp. 191–205.

[11] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2013, pp. 48–62.

[12] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Comput. Commun. Secur.*, 2005, pp. 340–353.

[13] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. Usenix Secur. Symp.*, 2013, pp. 1–17.

[14] C. Zhang *et al.*, "Practical control flow integrity and randomization for binary executables," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2013, pp. 559–573.

[15] C. Tice *et al.*, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. USENIX Secur. Symp.*, 2014, pp. 1–26.

[16] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2014, pp. 575–589.

[17] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proc. USENIX Secur. Symp.*, 2014, pp. 1–16.

[18] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proc. USENIX Secur. Symp.*, 2014, pp. 1–17.

[19] M. Conti *et al.*, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 952–963.

[20] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. USENIX Symp. Oper. Syst. Design Implement.*, 2014, pp. 1–18.

[21] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[22] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *Proc. 22nd SIGSAC ACM Conf. Comput. Commun. Secur.*, 2015, pp. 941–951.

[23] I. Evans *et al.*, "Missing the point(er): On the effectiveness of code pointer integrity," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2015, pp. 781–796.

[24] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song. (2015). *Poster: Getting the Point (er): On the Feasibility of Attacks on Code-Pointer Integrity*. Accessed: Jul. 6, 2017. [Online]. Available: http://www.ieee-security.org/TC/SP2015/posters/paper_48.pdf

[25] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight kernel protection against return-to-user attacks," in *Proc. USENIX Secur. Symp.*, 2012, pp. 1–16.

[26] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2010, pp. 380–395.

[27] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2014, pp. 292–307.

[28] J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang, "Comprehensive and efficient protection of kernel control data," *IEEE Trans. Inf. Forensics Security*, vol. 6, no. 4, pp. 1404–1417, Dec. 2011.

[29] *QEMU-the FAST! Processor Emulator*. Accessed: Jul. 6, 2017. [Online]. Available: http://www.qemu.org/

[30] *The LLVM Compiler Infrastructure*. Accessed: Jul. 6, 2017. [Online]. Available: http://llvm.org/

[31] *LLVM Language Reference Manual*. Accessed: Jul. 6, 2017. [Online]. Available: http://llvm.org/docs/LangRef.html#getelementptr-instruction

[32] *The LLVMLinux Project*. Accessed: Jul. 6, 2017. [Online]. Available: http://llvm.linuxfoundation.org/index.php/Main_Page

[33] *Kernel-based Virtual Machine*. Accessed: Jul. 6, 2017. [Online]. Available: https://www.linux-kvm.org/

[34] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proc. Int. Symp. Recent Adv. Intrusions Defenses*, 2008, pp. 1–20.

[35] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. ACM Symp. Oper. Syst. Principles*, 2007, pp. 1–17.

[36] J. Salwan. *Ropgadget*. Accessed: Jul. 6, 2017. [Online]. Available: http://shell-storm.org/project/ROPgadget/

[37] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "Ripe: Runtime intrusion prevention evaluator," in *Proc. 27th Annu. Comput. Secur. Appl. Conf.*, 2011, pp. 41–50.

[38] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Mar. 2016, pp. 179–194.

[39] *Supervisor Mode Access Prevention*. Accessed: Jul. 6, 2017. [Online]. Available: http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/

[40] *Supervisor Mode Access Prevention*. Accessed: Jul. 6, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Supervisor_Mode_Access_Prevention

[41] *CVE-2017-2636: Exploit the Race Condition in the n_hdlc Linux Kernel Driver Bypassing SMEP*. Accessed: Jul. 6, 2017. [Online]. Available: https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html

[42] *Practical Smep/Smap Bypass Techniques on Linux*. Accessed: Jul. 6, 2017. [Online]. Available: http://www.syscan360.org/slides/2016_SG_Vitaly_Nikolenko_Practical_SMEP_Bypass_Techniques.pdf

[43] *Phoronix Test Suite*. Accessed: Jul. 6, 2017. [Online]. Available: http://www.phoronix-test-suite.com/

[44] *LMbench—Tools for Performance Analysis*. Accessed: Jul. 6, 2017. [Online]. Available: http://www.bitmover.com/lmbench/lmbench3.tar.gz

[45] *UnixBench*. Accessed: Jul. 6, 2017. [Online]. Available: http://soft.vpser.net/test/unixbench/unixbench-5.1.2.tar.gz

[46] *SPEC CPU 2006*. Accessed: Jul. 6, 2017. [Online]. Available: http://www.spec.org/cpu2006/

[47] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 545–554.

[48] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering persistent kernel rootkits through systematic hook discovery," in *Proc. Int. Workshop Recent Adv Intrusion Detection*, 2008, pp. 21–38.

[49] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, "Exploiting and protecting dynamic code generation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[50] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proc. USENIX Secur. Symp.*, 2005, pp. 177–191.

[51] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proc. USENIX Symp. Oper. Syst. Design Implement.*, 2006, pp. 75–88.

[52] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.

[53] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-Rand: Practical mitigation of data-only attacks against page tables," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[54] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2011, pp. 353–362.

[55] B. Zeng, G. Tan, and U. Erlingsson, "Strato: A retargetable framework for low-level inlined-reference monitors," in *Proc. USENIX Secur. Symp.*, 2013, pp. 1–15.

[56] B. Niu and G. Tan, "Monitor integrity protection with space efficiency and separate compilation," in *Proc. SIGSAC ACM Conf. Comput. Commun. Secur.*, 2013, pp. 199–210.

[57] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proc. SIGSAC ACM Conf. Comput. Commun. Secur.*, 2015, pp. 914–926.

[58] V. van der Veen *et al.*, "Practical context-sensitive CFI," in *Proc. SIGSAC ACM Conf. Comput. Commun. Secur.*, 2015, pp. 927–940.

[59] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2008, pp. 233–247.

[60] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with return-less kernels," in *Proc. 5th ACM Eur. Conf. Comput. Syst.*, 2010, pp. 195–208.

[61] N. L. Petroni, Jr., and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 103–115.

[62] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtualarchitecture: A safe execution environment for commodity operating systems," in *Proc. ACM Symp. Oper. Syst. Principles*, 2007, pp. 1–16.

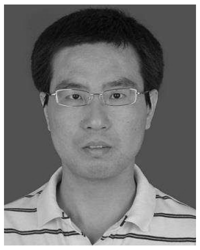
[63] *Rap: Rip Rop*. Accessed: Jul. 6, 2017. [Online]. Available: https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf

[64] G. Klein *et al.*, "seL4: Formal verification of an OS kernel," in *Proc. ACM Symp. Oper. Syst. Principles*, 2009, pp. 207–220.

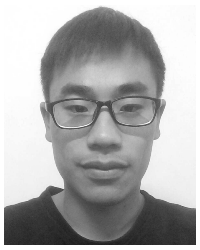
[65] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proc. USENIX Secur. Symp.*, 2015, pp. 1–17.

[66] I. Evans *et al.*, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2015, pp. 901–913.

[67] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2013, pp. 721–732.



Jinku Li received the B.S., M.S., and Ph.D. degrees in computer science from Xi'an Jiaotong University, Xi'an, China, in 1998, 2001, and 2005, respectively. From 2009 to 2011, he was a Post-Doctoral Fellow with the Department of Computer Science, North Carolina State University, Raleigh, NC, USA. He is currently an Associate Professor with the School of Cyber Engineering, Xidian University, Xi'an. His research focuses on system and mobile security.



Xiaomeng Tong received the B.S. degree in computer science from Xidian University, Xi'an, China, in 2015, where he is currently pursuing the M.S. degree with the School of Cyber Engineering. His research interests include operating system security and malware defense.

Fengwei Zhang received the Ph.D. degree in computer science from George Mason University in 2015. He is currently an Assistant Professor with the Computer Science Department, Wayne State University. His research interests are in the areas of systems security, with a focus on trustworthy execution, transparent malware debugging, transportation security, and plausible deniability encryption. He was a recipient of the Distinguished Paper Award in ACSAC 2017.

Jianfeng Ma received the Ph.D. degree in computer software and communications engineering from Xidian University, Xi'an, China, in 1995. From 1999 to 2001, he was with the Nanyang Technological University of Singapore as a Research Fellow. He is currently a Professor with the School of Cyber Engineering, Xidian University. His current research interests focus on information and network security.