Response to Reviewers on "FINE-CFI: Fine-grained Control-Flow Integrity for Operating System Kernels"

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

jkli@xidian.edu.cn, txmxidian@gmail.com, fengwei@wayne.edu, jfma@mail.xidian.edu.cn

We would like to deeply thank the anonymous reviewers and the Associate Editor for their timely, constructive comments. Your reviews provide valuable and unique insight that we have taken into great consideration to make this revision possible. In this document, please find our summarized revisions as well as detailed response to the reviewers' comments.

1 Revision Summary

Based on the reviews' comments, we have made major revisions in the following three aspects. And we have also uploaded a PDF **diff** file to the ScholarOne system for the reviewers.

- First, we have modified the prototype of FINE-CFI to prevent attacks by corrupting only the CS in the interrupt context.

  In the new prototype, we provide protection for the CS as well as the IP in the context information when an interrupt occurs. To this end, we need to instrument the *iret* instruction to verify the CS/IP pair for the interrupt return. Specifically, we insert seven instructions before *iret* instruction in function *irq_return()*. We use two *mov* instructions (i.e., *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. A new instrumentation example for interrupts is shown in Figure 5 (see Section III.C, pp.9).

  We have re-evaluated the performance for the new prototype, and the results show that the overhead is less than 10% on average. The details can be found in Section IV.B, pp.11-13.

- Second, the evaluation has been greatly expanded in the revision.

  In the security evaluation of our system, we have figured out the remaining gadgets in the original and instrumented Linux kernel using the open-source tool ROPgadget. As a result, ROPgadget found 44,063 gadgets in the original Linux kernel and 841 gadgets in the instrumented Linux kernel. We checked all these 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 could be leveraged to launch an effective ROP attack. The details can be found in Section IV.A, pp.11.

  In the performance evaluation of our system, we have introduced 3 new tests to our system. (1) In order to further measure the impact on real-world applications introduced by FINE-CFI, we have selected five representative real-world applications in 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 of our system. (2) 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. (3) 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 show that the performance overhead is less than 10% on average. The details can be found in Section IV.B, pp.11-13.

- Third, we have revised Section II.B carefully to address a few concerns raised by the reviewers.

  (1) We have provided a formal definition of *struct location vector*. In particular, we define the *struct location vector* of a function pointer as 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* (see Section II.B, pp.4).

  (2) We have introduced a more high-level description of our pointer analysis with a basic worklist algorithm that does not require compiler background knowledge to understand (see Algorithm 1, pp.4). After that, to describe our algorithm more intuitively, we leverage a simple example program to present the process to determine the point-to sets of indirect function calls (see Figure 1 and 2, pp.5).

  (3) We have made our main innovation clearer in the revision. Specifically, the main innovation of our retrofitted point-to analysis is that we introduce a new vector, called *struct location vector*, to infer the targets for 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 Section II.B, pp.4).

Next, we present our detailed response to individual reviewer's comments.

2 Response to Reviewer One

This paper proposes a novel CFI (Control Flow Integrity) approach for operating system kernels, named FINE-CFI. FINE-CFI uses a static analysis approach to determine a fine-grained CFG (Control Flow Graph) of the kernel, then enforces control flow to stay within that CFG. Moreover, it offers a hypervisor-based protection against corruption of interrupt context data.

The work is certainly of good quality. Most importantly, it combines fine granularity with low overhead, which is a significant improvement over previous work.

Authors response: Thanks for the summary and favorable comments.

However, there is a major issue that needs to be addressed. Through the paper, the authors state that the hypervisor-based protection covers the CS/IP pair in the interrupt context. However, the final prototype only covers IP (and this is indicated in just a footnote). Moreover, not protecting CS can have a serious security impact: it can allow an attacker to return from an interrupt to user code (IP, not corrupted) with kernel privileges (by corrupting the CPL in CS). This clearly only applies when SMEP is not present, but FINE-CFI makes no assumptions on SMEP. Since leaving CS unprotected

has an actual impact, I believe the prototype should be modified to include it and the performance re-evaluated. Evaluation of the main CFI mechanism is not affected.

Authors response: Thank you very much for the great comments! We have followed your suggestion and modified the prototype to prevent attacks by corrupting only the CS. In the new prototype, we provide protection for the CS as well as the IP. To achieve that, we need to instrument the *iret* instruction to verify the CS/IP pair for the interrupt return. Specifically, we insert seven instructions before *iret* instruction in function *irq_return()*. We use two *mov* instructions (i.e., *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. A new instrumentation example for interrupts is shown in Figure 5 (see Section III.C, pp.9).

We have re-evaluated the performance for the new prototype, and the results show that the overhead is less than 10% on average. As well, we have introduced several other benchmarks to make further evaluation for our system. The details can be found in Section IV.B, pp.11-13.

There are also other improvements I would like to see:
1) Expand and clarify on ret2usr attacks. The classic ret2usr involves corrupting non-interrupt control data (e.g., a kernel function pointer), and kernel CFI mitigates it. The kind of ret2usr that requires hypervisor protection involves corrupting the interrupt context and is not covered by the provided reference ([25]). I would also appreciate if you touched upon SMEP and SMAP, as they are effective hardware countermeasures to classic ret2usr.

Authors response: Thanks for your suggestion. We have made expansion and clarification on the two kinds of ret2usr attack in the revision (see Section I, pp.2). As pointed out by you, the classic ret2usr attacks only corrupt non-interrupt (usual) control data in kernel space, e.g., a kernel function pointer or return address. However, the novel ret2usr attacks could corrupt IP and CS in the interrupt context to transfer execution control-flow to the user-space code controlled by attackers with kernel privileges. As a result, we also need to provide protection for the control data in the interrupt context, as well as the usual control data in kernel space.

At the same time, in the revision, we have presented several feasible attacks to bypass SMEP/SMAP, and explained why all of them can be defeated by FINE-CFI (see Section IV.A, pp.11). Specifically, it has been showed that SMEP/SMAP could be bypassed with kernel 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 [1]. Nikolenko shows another trick to disable SMEP by combining several useful kernel gadgets and pivoting the stack for clearing the bit, or bypass SMAP by abusing vDSO [2]. However, FINE-CFI prevents both the kernel code-reuse attacks and the ret2usr attacks, by providing fine-grained CFI protection and the control data protection in the interrupt context. Thus, FINE-CFI is immune to the above attacks.

[1] "Cve-2017-2636: exploit the race condition in the n_hdlc linux kernel driver bypassing smep." https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html.

[2] "Practical smep/smap bypass techniques on linux." http://www.syscan360.org/slides/2016_SG_Vitaly_Nikolenko_Practical_SMEP_Bypass_Techniques.pdf.

2) Clarify the instrumentation of direct calls. In Section III.B the authors mention the replacement of 50,677 direct calls. Why are they replaced, when the integrity of kernel code is assumed in Section II.A?

Authors response: Indeed, the reason why it instruments direct call instructions is due to the adoption of *indexed hooks* approach to enforce CFI protection.

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. For each ret instruction, its jump table (i.e., return table) contains all the return addresses it may return to according to the CFG. With that, *indexed hooks* leverages the extended compiler *LLVM* to instrument all the return address-related instructions by replacing the return address with a return table index. In the high level, to provide protection for return addresses, a "call dst" instruction is instrumented into "push $ret_table_index; jmp dst", and a "ret" instruction is instrumented into "pop %reg; jmp *RetTable(%reg)". We have made clarification in the revision (see Section III.B, pp.8).

3) Explicitly mention the performance numbers for other kernel CFI implementations when comparing them to FINE-CFI, as not all readers are familiar with them.

Authors response: Thanks for your suggestion. In the revision, we have explicitly compared the performance overhead of KCoFI and [1] to FINE-CFI. The results indicate that the performance overhead of FINE-CFI is smaller than the other two systems (see Section IV.B, pp.11-12).

Specifically, in the Phoronix Test Suite, for postmark benchmark which is used to measure the file system performance, FINE-CFI only incurs 7.69% overhead while KCoFI incurs 1.96x on average. At the same time, in the evaluation results of LMbench, FINE-CFI's maximum overhead for fork+exit operation is only 11.05% while KCoFI incurs 3.50x overhead on average. On the other hand, in the evaluation results of UnixBench, FINE-CFI incurs 7.20% overhead for a Linux kernel, while [1] incurs 11.91% overhead for a FreeBSD kernel.

[1] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in Security and Privacy (EuroS&P), 2016 IEEE European Symposium on, pp.179-194, IEEE, 2016.

4) Define the struct location vector more formally. Right now it is defined through examples, and while it can be understood it would be better to have an explicit definition of it, especially because there can be multiple indexes for nested fields, which is not immediately clear from Section II.B.

Authors response: Thanks for your suggestion. In the revision, we have provided the formal definition of *struct location vector* (see Section II.B, pp.4). In particular, we define the *struct location vector* of a function pointer as 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, ...)* (*ele_order* denotes the element' order number in a *%strut.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 *%struc.sb* struct, which is the 0*th* element in a *%struct.sb*-typed array.

5) In Figure 1(b), there seems to be an omission that is not indicated: the initialization (presumably from a stack allocation) of %1 and %2 is not present. While not relevant for the analysis, the omission should be indicated.

Authors response: Thanks a lot for your correction. We have indicated the omission in the revision (see Section II.B, pp.4). As you pointed out, %1 and %2 are indeed two temporary variables from stack allocation and we omit them.

6) Proofread the paper more thoroughly, as there are several grammatical mistakes and non-idiomatic wordings. There is also a typo in Section III.A: "[...] %18 has been cas(T)ed from [...]".

Authors response: Thanks for your correction and suggestion. We have corrected the typo in the revision. And we have reviewed the revision carefully several times and fixed some mistakes.

As a final note, the hypervisor-based protection actually does more than thwarting ret2usr attacks. It also protects against kernel code reuse, which is a common SMEP bypass.

Authors response: Thanks for your information. We agree with your opinion. Indeed, in the revision, we have leveraged several SMEP/SMAP bypass attacks to clarify that our hypervisor-based protection can also protect against kernel code-reuse attacks (see Section IV.A, pp.11).

3 Response to Reviewer Two

Strengths:

- Prototype implementation

- Good performance

Weaknesses:

- Contribution not clear

- Explanation of the idea requires revision

- Security analysis not convincing

Authors response: Thanks for the summary. We will respond to your concerns in the following.

Modern kernels are written in unsafe languages like C/C++, and prone to memory corruption vulnerabilities. Attackers exploit these vulnerabilities to overwrite code pointers, which are afterwards used by the application, to hijack the control flow, and get full control of the system. One prominent defense against control-flow hijacking attacks is control-flow integrity (CFI) which verifies each code pointer before it is used as a branch target. This paper presents the design and implementation of CFI for operating system kernels, called FINE-CFI. In contrast to user mode CFI, kernel CFI comes with additional challenges as the kernel makes heavy use of single code pointers in various data structures.

In general, the presented design makes a solid impression. FINE-CFI is a compiler extension that instruments the kernel to enforce CFI checks before each indirect branch. Further, FINE-CFI utilizes a trusted hypervisor to protect the context data of interrupts. This is important because the interrupt context contains the saved instruction pointer. The performance evaluation indicates an acceptable average performance overhead of less than 5%.

Authors response: Thanks for the summary and comments.

The main issue of the paper is that all the previous described design of FINE-CFI is a retrofitted version of the authors previous work 'Comprehensive and efficient protection of kernel control data' [27]. Hence, it is a bit confusing that large parts of the paper are dedicated to explaining and evaluating this design.

Authors response: Thanks for your patience. We fully understand what you're confusing and we do borrow the instrumentation scheme *indexed hooks* in our previous work 'Comprehensive and efficient protection of kernel control data'. However, as mentioned by you in the following, the main contribution of this paper is that it leverages a retrofitted context-sensitive and field-sensitive pointer analysis (rather than the dynamic profiling approach in the previous work) 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. 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 leverage *indexed hooks* rather than the original label-based approach as *indexed hooks* solved the destination equivalence problem, which could possibly lead to coarse-grained CFI.

We have made a further clarification in the revision (see Section VI, pp.14).

The actual contribution of the paper is the difference in inferring the policy that is enforced by CFI runtime checks. The previous work [27] used a dynamic analysis while this paper uses a compiler-based approach. In particular, this paper bases their policy on function signatures (return type, number and type of arguments). Unfortunately, this is a known technique and published [a] (slide 23) which weakens

the scientific contribution. However, this could be compensated by providing a very strong evaluation and documentation.

[a] https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf

Authors response: Thank you. We agree that the function signature-based policy is not a novel idea, though it is not an easy task to implement such a policy in a large code base. Further, we want to clarify that our pointer analysis approach is not merely based on function signatures. Indeed, to improve the precision of the function pointer's point-to set, we introduce a new vector, i.e., *struct location vector*, to infer the targets for indirect function invocations. 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. 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 Section II.B, pp.4). Additionally, as you suggested in the following, we have performed a further security evaluation using a ROPgadget tool to find and analyze the remaining gadgets after our instrumentation, and the results show that none of these gadgets could be used to launch an effective ROP attack (see Section IV.A, pp.11).

This leads to the second shortcoming of the paper. The paper would greatly benefit from revising Section II.B. While such a detailed explanation of the used techniques is desirable for the implementation section, the authors should also include a more high-level description of their technique that does not require compiler background knowledge.

Authors response: Thank you very much and it is a good point. In the revision, we have followed your suggestion and revised Section II.B carefully.

First, we have provided a formal definition of *struct location vector*. In particular, we define the *struct location vector* of a function pointer as 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, ...)* (*ele_order* denotes the element' order number in a *%strut.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; $i >= 1$) (see Section II.B, pp.4).

Second, we have introduced a more high-level description of our pointer analysis with a basic worklist algorithm that does not require compiler background knowledge to understand (see Algorithm 1, pp.4). After that, to describe our algorithm more intuitively, we leverage a simple example program to present the process to determine the point-to sets of indirect function calls (see Figure 1 and 2, pp.5).

Third, we have further made our main conceptual ideas and innovations clearer in the revision (see Section II.B, pp.4).

For a strong evaluation, there are couple of issues which need to be addressed:
1) It was shown and the community agreed that the AIR metric is not very useful [57]. Hence, the paper would greatly benefit from an (semi-)automatic analysis of the remaining attack surface which means looking at the remaining gadgets and if they could be used in an attack.

Authors response: Thank you very much for the constructive comment. Following your suggestion, we have figured out the remaining gadgets in the original and instrumented Linux kernel using the open-source tool ROPgadget. As a result, ROPgadget found $44,063$ gadgets in the original Linux kernel and 841 gadgets in the instrumented Linux kernel. We checked all these 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 could be leveraged to launch an effective ROP attack (see Section IV.A, pp.11).

2) The choice of benchmarking test is not always clear: CPU heavy user mode benchmark (SPEC CPU)

are expected to not have any serious performance degradation when kernel CFI is applied. Further, it is not clear how single arithmetic operations are meaningful in this context. Could FINE-CFI be applied to the SPECCPU benchmarks?

Authors response: Thanks for your comments. We also notice that it seems to be no serious performance overhead on SPEC CPU benchmark when kernel CFI is applied. To further measure the performance overhead imposed by our approach, we have followed your suggestion and applied FINE-CFI to four benchmarks in SPEC CPU2006 benchmark suite, i.e., 401.bzip2, 403.gcc, 429.mcf, and 458.sjeng. These benchmarks depend on the *libc* library as well as some objects in C runtime library (e.g., crt1.o, crti.o, crtn.o). To achieve that, we need to instrument the benchmark programs as well as the dependencies. The evaluation results show that the performance overheads introduced by 401.bzip12, 403.gcc, 429.mcf, and 458.sjeng are 7.18%, 4.13%, 12.03%, and 10.12%, respectively. The details can be found in Table V, pp.12.

On the other hand, it seems to be no any performance overhead on the arithmetic operations in LMbench. The possible reason is that these operations might not go through enough instrumented sites in the kernel. We have removed the single arithmetic operations in LMbench evaluation in the revision (see Figure 8, pp.12).

How about testing the impact on real-world applications with the help of the Phoronix benchmarking suite?

Authors response: Thanks for your suggestion. In the revision, we have selected 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 our approach. The results show that the overhead is acceptable (5.39% for *New-f* and 9.89% for *New-f+i* on average, respectively). The details can be found in Table II, pp.12.

Is it possible break the results further down to see the impact on the forward/backward edge protection?

Authors response: This is a good point. As mentioned earlier, we have applied our CFI enforcement to four benchmarks (i.e., 401.bzip2, 403.gcc, 429.mcf, and 458.sjeng) in the SPEC CPU2006 benchmark suite. At the same time, we also achieve 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 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. The results can be found in Table V, pp.12.

It would be nice to see a discussion on the impact of the call/ret stack.

Authors response: Thanks for your suggestion. In the revision, we have raised the discussion in Section V, pp.13. 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.

Nit Picks:

P2: verifies those information $->$ that

P5: So we put $->$ So, we put

P12: thus it introduces $->$ thus, it introduces

Authors response: Thank you very much for your correction. We have fixed the mistakes in the revision.

4 Response to Reviewer Three

This paper presents fine-grained control-flow integrity (FINE-CFI) for operating system kernel code. FINE-CFI is based on a static analysis and instrumentation LLVM pass to identify the set of allowed branch targets for each indirect branch and so-called indexed hooks to enforce CFI checks at run-time. In addition, it maintains an interrupt stack to detect attacks that overwrite return addresses during a context switch.

Authors response: Thank you for the summary.

Memory corruption attacks hijack the program's control flow to induce malicious program actions. Control-flow integrity (CFI) mitigates these attacks by enforcing that the program's control flow follows a legitimate path in the program's pre-defined control-flow graph. However, existing schemes are either coarse-grained or do not target kernel code. This paper aims to tackle these limitations by developing FINE-CFI. Given that most work in this area target user-land code, I appreciate that this paper targets instead the kernel code. However, the paper does not describe the challenges when applying CFI to kernel code. One gets the impression that this is mostly an engineering task rather than developing any new mechanisms. As such, I fear that the paper's main contribution is rather limited and incremental.

Authors response: Thank you very much for the constructive comments. In the revision, we have followed your suggestion and presented three additional challenges when applying CFI in kernels (the details can be found in Section I, pp.2).

The first challenge is due to the asynchronous nature of context switching and interrupt handling in kernel space. 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 requires 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.

In general, the paper is well-written and attempts to describe the low-level implementation details based on example code. On the other hand, the main conceptual ideas and innovations are not really described.

Authors response: Thanks for the comments and criticism. We have revised Section II.B carefully and made our main conceptual ideas and innovations clearer in the revision.

First, we have provided a formal definition of *struct location vector*. In particular, we define the *struct location vector* of a function pointer as the location of the function pointer member in a (nested) struct (see Section II.B, pp.4).

Second, we have introduced a more high-level description of our pointer analysis with a basic worklist algorithm that does not require compiler background knowledge to understand (see Algorithm 1, pp.4). After that, to describe our algorithm more intuitively, we leverage a simple example program to present the process to determine the point-to sets of indirect function calls (see Figure 1 and 2, pp.5).

Third, we have described our main innovation clearly in the revision. Specifically, the main innovation of our retrofitted point-to analysis is that we introduce a new vector, called *struct location vector*, to infer the targets for 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 Section II.B, pp.4).

The presented approach for points-to analysis seems reasonable to me and the evaluation demonstrates a significant reduction of the target branch set. Unfortunately, the paper does not really compare the target branch set to previous work in this area.

Authors response: Thanks a lot. In the revision, we have provided the comparison of the reduction of the target branch set to three previous work in this area. The results show the AIR metric of FINE-CFI (99.998%) is better than bin-CFI (98.86%), KCoFI (98.18%) and Ge's work [1] (99.6% for return targets and 99.1% for indirect call/jmp targets). The details can be found in Section IV.A, pp.10.

[1] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in Security and Privacy (EuroS&P), 2016 IEEE European Symposium on, pp.179-194, IEEE, 2016.

The modified interrupt handling to protect return addresses used in context switches leverages a shadow stack like approach. I am unsure about the novelty of this approach. Further, I miss a justification that leveraging a hypervisor to protect the interrupt handling resembles a reasonable approach.

Authors response: Yes, we do leverage a shadow stack like approach (called interrupt stack) to protect control data in the interrupt context. As a usual approach, a number of systems use it to provide protection for return addresses pushed on the stack by function calls. However, the difference between shadow stack approach and our protection mechanism is that interrupts could occur at any time or any valid instruction boundary, but function calls occur at the fixed instruction boundaries. In other words, it can be predetermined when and where an return address will be pushed on the stack (i.e., when a call instruction is being executed), so that the shadow stack approach can protect it by instrumenting all the call (and ret) instructions. However, kernel itself cannot predetermine when and where an interrupt may occur and it is impossible for us to do that by instrumenting every instruction in the kernel. As a result, we propose the hypervisor-based approach to protect control data in the interrupt context, as the hypervisor can take over the guest VM when an interrupt happens. So that we can backup the control data in the hypervisor for later verification when returning from the interrupt handler.

Further, we want to clarify that the main novelty of this paper is that it leverages a retrofitted context-sensitive and field-sensitive pointer analysis to obtain the fine-grained CFG of the kernel, which greatly improves the precision of the result. In particular, we introduce a new vector, i.e., struct location vector, to infer the targets for each indirect invocation. It largely reduces the number of targets of indirect call/jmp instructions (average 13.14 for each indirect call/jmp instruction).

The evaluation is based on several well-known benchmark programs and demonstrates the overhead is acceptable. The security evaluation also includes tests against existing exploit payloads. FINE-CFI is able to prevent the exploits of the RIPE testbed.

Authors response: Thanks for the summary. We have added several new security and performance evaluation results in the revision (see Section IV, pp.10-13).

I miss any comparison to a recent, very closely related work on kernel-CFI: "Fine-Grained Control-Flow Integrity for Kernel Software" (EURO SP 2016).

Authors response: Thank you very much for the information. We have done a comparison to that recent work (i.e., "Fine-Grained Control-Flow Integrity for Kernel Software") in the revision (see Section VI, pp.14). 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 the 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. As a result, we achieve 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). We also compare the performance to that work in our revision (see Section IV.B, pp.12). The results show that the performance overhead incurred by FINE-CFI is smaller than that work.