KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels

John Criswell, Nathan Dautenhahn, and Vikram Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {criswell,dautenh1,vadve}@illinois.edu

Abstract

We present a new system, KCoFI, that is the first we know of to provide complete Control-Flow Integrity protection for commodity operating systems without using heavyweight complete memory safety. Unlike previous systems, KCoFI protects commodity operating systems from classical control-flow hijack attacks, return-to-user attacks, and code segment modification attacks. We formally verify a subset of KCoFI's design by modeling several features in small-step semantics and providing a partial proof that the semantics maintain control-flow integrity. The model and proof account for operations such as page table management, trap handlers, context switching, and signal delivery. Our evaluation shows that KCoFI prevents *all the gadgets* found by an open-source Return Oriented Programming (ROP) gadget-finding tool in the FreeBSD kernel from being used; it also reduces the number of indirect control-flow targets by 98.18%. Our evaluation also shows that the performance impact of KCoFI on web server bandwidth is negligible while file transfer bandwidth using OpenSSH is reduced by an average of 13%, and at worst 27%, across a wide range of file sizes. *PostMark*, an extremely file-system intensive benchmark, shows 2x overhead. Where comparable numbers are available, the overheads of KCoFI are far lower than heavyweight memory-safety techniques.

I. INTRODUCTION

Despite much research, memory safety attacks are still a scourge for C/C++ programmers. Worse yet, most commodity operating systems are written in C/C++ and are therefore susceptible to memory safety attacks. As the operating system (OS) is part of the Trusted Computing Base (TCB) in nearly all commodity systems, a vulnerability in the OS can undermine the security of an entire system.

Many memory safety attacks work by diverting a program's control flow to instructions of the attackers choosing; these instructions may be injected by the attacker [1] or may already exist within the program [2], [3]. Control-flow integrity (CFI) is a family of security policies that thwart such attacks. Traditional CFI requires that all computed branches (e.g., returns from functions and indirect function calls) jump to virtual addresses that are designated as correct via static analysis [4]. Additional restrictions to CFI [5], [6] require that the instructions do not change.

Enforcing CFI on commodity operating system kernel code could provide protection against control hijack attacks

that is comprehensive, efficient, and straightforward to implement. However, operating systems pose three challenges for existing CFI techniques. First, not all targets of indirect control transfers can be determined statically from the kernel code. Interrupts can occur at *any* instruction boundary, so the kernel must be able to transfer control to any interrupted instruction on a return from interrupt. Second, operating systems operations affect control flow in complicated ways. Signal handler dispatch, for example, modifies the program counter in interrupted program state saved in memory [7], [8], and efficient user-kernel memory copying functions modify interrupted kernel state [7] to recover from page protection faults. Third, operating systems have access to privileged hardware that invalidate assumptions commonly made by CFI techniques. As an example, some CFI systems [4], [9] assume that the code segment is non-writable. Errant DMA and MMU configurations can invalidate that assumption [5], [6].

Most solutions for enforcing CFI [4], [10], [9] do not protect commodity operating system code. The few that do protect system-level code have serious limitations: Hyper-Safe [6] only protects a hypervisor and does not provide control-flow integrity for operations found in operating systems (e.g., signal handler dispatch); it also does not protect against return to user (ret2usr) attacks [11] that corrupt the program counter saved on interrupts, traps, and system calls to execute code belonging to less-privileged software. The kGuard [11] system, designed to thwart ret2usr attacks, enforces a very weak CFI variant that only ensures that control-flow is directed at virtual addresses within the kernel; some of its protection is probabilistic, and it does not handle attacks that use the MMU to change the instructions within the code segment. Secure Virtual Architecture (SVA) [12], [5] provides comprehensive protection against control hijacking attacks, but it does so with heavyweight memory-safety techniques that have relatively high overheads even after being optimized by techniques using sophisticated whole-program pointer analysis [13].

Furthermore, only the original CFI system [4] formally proves that it enforces control-flow integrity, and it does not model features such as virtual memory, trap handlers, context switching, and signal delivery found in modern operating systems. Having an approach for enforcing control-flow integrity on these operations that has been formally verified would increase confidence that the approach works correctly.

We have built a system named KCoFI (Kernel Control Flow Integrity, pronounced "coffee") that aims to provide comprehensive, efficient, and simple protection against control flow attacks for a complete commodity operating system. KCoFI

operates between the software stack and processor. Essentially, KCoFI uses traditional label-based protection for programmed indirect jumps [4] but adds a thin run-time layer linked into the OS that protects some key OS data structures like thread stacks and monitors all low-level state manipulations performed by the OS. Our system provides the first comprehensive control-flow integrity enforcement for commodity OS kernels that does not rely on slower and more sophisticated memory safety techniques. Our protection thwarts both classical control flow attacks as well as ret2usr attacks. To verify that our design correctly enforces control-flow integrity, we have built a formal model of key features of our system (including the new protections for OS operations) using small-step semantics and provided a partial proof that our design enforces control-flow integrity. The proofs are encoded in the Coq proof system and are mechanically verified by Coq.

The contributions of our system are as follows.

- We provide the first complete control-flow integrity solution for commodity operating systems that does not rely on sophisticated whole-program analysis or a much stronger and more expensive security policy like complete memory safety.

- We have built a formal model of kernel execution with small-step semantics that supports virtual to physical address translation, trap handling, context switching, and signal handler dispatch. We use the model to provide a partial proof that our design prevents CFI violations. (We do not verify our implementation.)

- We evaluate the security of our system for the FreeBSD 9.0 kernel on the x86-64 architecture. We find that all the Return Oriented Programming (ROP) gadgets found by the ROPGadget tool [14] become unusable as branch targets. We also find that our system reduces the average number of possible indirect branch targets by 98.18%.

- We evaluate the performance of our system and find that KCoFI has far lower overheads than SVA [12], [5], the only other system which provides full control-flow integrity to commodity OS kernels. Compared to an unmodified kernel, KCoFI has relatively low overheads for server benchmarks but higher overheads for an extremely file-system intensive benchmark.

The remainder of the paper is organized as follows: Section II describes our attack model. Section III provides an overview of the KCoFI architecture. Section IV presents the design of KCoFI and how it enforces control-flow integrity, and Section V presents an overview of our formal control-flow integrity proof. Section VI describes our implementation while Section VII evaluates its efficacy at thwarting attacks and Section VIII describes the performance of our system. Section IX describes related work, Section X describes future work, and Section XI concludes.

II. Attack Model

In our attack model, we assume that the OS is benign but may contain vulnerabilities; we also assume that the OS has been properly loaded without errors and is executing. Our model allows the attacker to trick the kernel into attempting to modify any memory location. We additionally assume that the attacker is using such corruption to modify control-data, including targets that are not of concern to traditional CFI techniques, e.g., processor state (including the PC and stack pointer) saved in memory after a context-switch; trap and interrupt handler tables; invalid pointer values in user-kernel copy operations; malicious MMU reconfiguration; etc. Non-control data attacks [15] are excluded from our model.

Notice that external attackers in our model can influence OS behavior only through system calls, I/O, and traps. For example, dynamically loaded device drivers are assumed not to be malicious, but may also be buggy (just like the rest of the OS kernel), and will be protected from external attack. We assume that the system is employing secure boot features such as those found in AEGIS [16] or UEFI [17] to ensure that KCoFI and the kernel are not corrupted on disk and are the first pieces of software loaded on boot. We further assume that the attacker does not have physical access to the machine; hardware-based attacks are outside the scope of our model.

III. KCoFI Infrastructure

KCoFI has several unique requirements. First, it must instrument commodity OS kernel code; existing CFI enforcement mechanisms use either compiler or binary instrumentation [4], [10], [18]. Second, KCoFI must understand how and when OS kernel code interacts with the hardware. For example, it must understand when the OS is modifying hardware page tables in order to prevent errors like writeable and executable memory. Third, KCoFI must be able to control modification of interrupted program state in order to prevent ret2usr attacks.

The Secure Virtual Architecture (SVA) [12], [5] provides the infrastructure that KCoFI needs. As Figure 1 shows, SVA interposes a compiler-based virtual machine between the hardware and the system software (such as an operating system or hypervisor). All software, including the operating system and/or hypervisor, is compiled to the virtual instruction set that SVA provides. The SVA virtual machine (VM) translates code from the virtual instruction set to the native instruction set either ahead-of-time (by caching virtual instruction set translations) or just-in-time while the application is running [19].
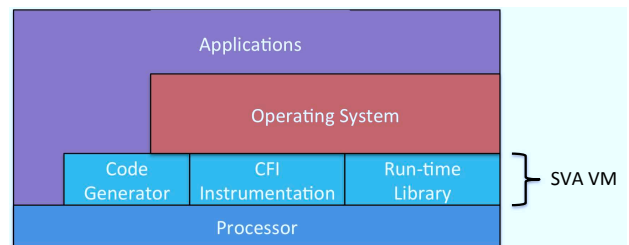

Fig. 1. SVA/KCoFI Architecture

The core SVA virtual instruction set is based on the LLVM compiler intermediate representation (IR) [20]. This instruction set is a RISC-like instruction set with an unlimited number of scalar registers in SSA form, making compiler analysis more effective and efficient than on binary code [20]. Furthermore, programs in LLVM IR can be serialized to disk as a self-contained object code format (called bitcode), permitting link-time, whole-program analysis and transformation, and also

allowing such programs to be shipped in LLVM bitcode form for analysis and transformation on the user's machine, e.g., at install time or run-time [20].

SVA adds a set of instructions to LLVM called SVA-OS, which replace the inline assembly code needed in commodity operating systems to communicate with the hardware and to do low-level state manipulation [19], [12], [5]. SVA-OS handles primitive operations such as context switching, signal handler dispatch, MMU configuration, and I/O reads and writes. Additionally, the design of the SVA-OS instructions permits the SVA VM to control their behavior to ensure that they do not violate any security policies that SVA enforces.

Because the operating system must interface with the hardware via the SVA-OS instructions, it must be ported to the SVA virtual instruction set. This is similar to porting the operating system to a new architecture, but a relatively simple virtual architecture, and only requires modifying the lowest-level parts of the kernel. No reorganization of the kernel or modifications to drivers are needed.

The SVA infrastructure enables KCoFI to enforce a CFI policy by using the SVA compiler instrumentation capabilities and using the SVA-OS instruction set to identify and control both OS kernel/hardware interactions and OS kernel/application interactions. KCoFI requires all OS code, including kernel extensions, to be compiled to the virtual instruction set but allows applications to be compiled to either the virtual or native instruction set.

IV. DESIGN

In this section, we describe the CFI policy that KCoFI enforces and the hardware and compiler mechanisms it employs to enforce the policy.

A. Control-flow Integrity Policy and Approach

KCoFI enforces context-insensitive CFI like that of Abadi et. al. [4]: calls to functions must jump to the beginning of some function, and returns must jump back to one of the call sites that could have called the exiting function. The return address is not itself protected, so it is possible for a function to dynamically return to a call site other than the one that called the function in that specific execution.

To enforce CFI, Abadi et. al. [4] insert special byte sequences called *labels* at the targets of indirect control transfers within the code segment. These labels must not appear anywhere else within the instruction stream. Their technique then inserts code before indirect jumps to check that the address that is the target of the indirect jump contains the appropriate label. Abadi et. al. provided a formal proof that their technique enforces control-flow integrity if the code segment is immutable [4].

The KCoFI VM instruments the code with the needed labels and run-time checks when translating code from the virtual instruction set to the processor's native instruction set. To avoid complicated static analysis, KCoFI does not attempt to compute a call graph of the kernel. Instead, it simply labels all targets of indirect control transfers with a single label. Our design also uses a jump table optimization [18] to reduce the number of labels and CFI checks inserted for

switch statements. While our current design effectively uses a very conservative call graph, note that a more sophisticated implementation that computes a more precise call graph can be made without changing the rest of the design. Also, the MMU protections (discussed in Section IV-C) ensure that the code segment is not modified by errant writes.

One issue with using CFI labels is that a malicious, native code user-space application could place CFI labels within its own code to trick the instrumentation into thinking that its code contains a valid kernel CFI target [11]. KCoFI solves this problem by adapting a technique from kGuard [11]; before checking a CFI label, it masks the upper bits of the address to force the address to be within the kernel's address space. This approach allows KCoFI to safely support legacy, native code applications that are not compiled to the virtual instruction set.

Similarly, the SVA-OS instructions described later in this section are implemented as a run-time library that is linked into the kernel. This run-time library is instrumented with a disjoint set of CFI labels for its internal functions and call sites to ensure that indirect branches in the kernel do not jump into the middle of the implementation of an SVA-OS instruction. In this way, the run-time checks that these library functions perform cannot be bypassed.

B. Protecting KCoFI Memory with Software Fault Isolation

The original CFI technique of Abadi et al. [4] is stateless in that the only data used are constant labels embedded in the code segment of the application being protected, either as immediate operands to checking instructions or as constant labels at control transfer targets.[1] KCoFI, however, needs to maintain some additional state to protect privileged kernel behaviors, which do not occur in userspace code. This state includes hardware trap vector tables, page mapping information, interrupted program state (as described in Section IV-F), and other state that, if corrupted, could violate control-flow integrity. While the MMU can protect code memory because such memory should never be writeable, KCoFI will need to store this state in memory that can be written by KCoFI but not by errant operating system and application writes. KCoFI uses lightweight instrumentation on kernel store instructions to protect this memory: essentially a version of software-fault isolation [21]. (An alternative would be to use MMU protections on KCoFI data pages as well, but that would incur additional numerous TLB flushes.)

As Figure 2 shows, our design calls for a reserved portion of the address space called KCoFI memory which will contain the KCoFI VM's internal memory. KCoFI uses the MMU to prevent user-space code from writing into kCoFI memory. To prevent access by the kernel, KCoFI instruments all instructions in the kernel that write to memory; this instrumentation uses simple bit-masking that moves pointers that point into KCoFI memory into a reserved region of the address space (marked "Reserved" in Figure 2). This reserved region can either be left unmapped so that errant writes are detected and reported, or it can have a single physical frame mapped to every virtual page within the region so that errant writes are silently ignored. Note that only stores need instrumentation;

---

[1]One variant of their design uses x86 segmentation registers to protect application stack frames containing return addresses.

TABLE I.   KCoFI MMU Instructions

Name	Description
sva.declare.ptp (void * ptp, unsigned level)	Zeroes the physical page mapped to the direct map pointer *ptp* and marks it as a page table page at level *level*.
sva.remove.ptp (void * ptp)	Checks that the physical page pointed to by direct map pointer *ptp* is no longer used and marks it as a regular page.
sva.update.l1.mapping (void * ptp, unsigned trans)	If *ptp* is a direct map pointer to an L1 page, validate that the translation *trans* does not violate any security policies and store it into *ptp*.
sva.update.l2.mapping (void * ptp, unsigned trans)	If *ptp* is a direct map pointer to an L2 page, validate that the translation *trans* maps an L1 page and store *trans* into *ptp*.
sva.update.l3.mapping (void * ptp, unsigned trans)	If *ptp* is a direct map pointer to an L3 page, validate that the translation *trans* maps an L2 page and store *trans* into *ptp*.
sva.update.l4.mapping (void * ptp, unsigned trans)	If *ptp* is a direct map pointer to an L4 page, validate that the translation *trans* maps an L3 page and store *trans* into *ptp*.
sva.load.pagetable (void * ptp)	Check that the physical page mapped to the direct map pointer *ptp* is an L4 page and, if so, make it the active page table.

none of the KCoFI internal data needs to be hidden from the kernel, and, as a result, can be freely read by kernel code.

Fig. 2.   KCoFI Address Space Organization

C. MMU Restrictions

As previous work has shown [5], [6], MMU configuration errors can lead to violations of security policies enforced by inline reference monitors. As KCoFI's CFI mechanism must keep code read-only and the store instrumentation makes assumptions about the address space layout, KCoFI must be able to restrict how the operating system configures the MMU.

The SVA infrastructure forces hardware page table pages to be read-only and requires the OS to use special instructions to make changes to the page table pages [5]. KCoFI simplifies and enhances the original SVA-OS MMU instructions; these instructions are shown in Table I.

The KCoFI VM maintains a structure within its portion of the address space called the *direct map*. The direct map is a one-to-one mapping between virtual addresses within the direct map and physical frames of memory. All pages in the direct map will be read-only. The purpose of the direct map is to provide the KCoFI VM and the OS kernel with a known virtual address for every physical address. When the OS asks the KCoFI VM to make page table updates, it will identify page table pages by their direct map virtual address.

The `sva.declare.ptp()` instruction informs the KCoFI VM of which frames will be used for hardware page

table pages and at which level of the page table hierarchy these frames will be used. The KCoFI VM will only permit pages that are not in use to be declared for use as page table pages, and it will zero the page's contents to ensure that no stale virtual to physical page translations remain in the page. When the system wishes to stop using a frame as a page table page, it can call `sva.remove.ptp()`. When called, `sva.remove.ptp()` verifies that the frame is no longer referenced by any page table pages; if the check passes, it allows the frame to be mapped read/write into the virtual address space and used for kernel or user-space data.

The `sva.update.l<n>.mapping()` instructions write a new page table entry into a page table page previously declared using the `sva.declare.ptp()` instruction. The KCoFI VM will first vet the mapping before inserting it into the page table page at the specified offset. For example, if the mapping should insert an L2 page table page, the checks ensure that the physical address specified in the translation is a page previously declared as an L2 page. The instructions will also keep count of how many references there are to each physical page frame.

Switching from one page table to another is done by the `sva.load.pagetable()` instruction which requires that it be given the address of a level 4 page table page.

There are two ways in which reconfiguration of the MMU can allow the operating system to bypass the protections provided by the compiler instrumentation. First, an errant operating system may reconfigure the virtual addresses used for KCoFI memory or the code segment so that they either permit write access to read-only data or map new physical frames to the virtual pages, thereby modifying their contents. Second, an errant operating system might create new virtual page mappings in another part of the address space to physical pages that are mapped into KCoFI memory or the code segment. Since the CFI and store instrumentation makes assumptions about the location of the code segment and KCoFI memory within the address space, the MMU instructions must ensure that those assumptions hold. If these assumptions were to be broken, then both the code segment and KCoFI's internal data structures could be modified, violating control-flow integrity.

The KCoFI MMU instructions enforce the following restrictions on MMU configuration in order to protect the native code generated by the KCoFI VM:

TABLE II.   KCoFI Interrupt Context Instructions

Name	Description
sva.icontext.save (void)	Push a copy of the most recently created Interrupt Context on to the thread's Saved Interrupt Stack within the KCoFI VM internal memory.
sva.icontext.load (void)	Pop the most recently saved Interrupt Context from the thread's Saved Interrupt Context stack and use it to replace the most recently created Interrupt Context on the Interrupt Stack.
sva.ipush.function (int (*f)(...), ...)	Modify the state of the most recently created Interrupt Context so that function f has been called with the given arguments. Used for signal handler dispatch.
sva.init.icontext (void * stackp, unsigned len, int (*f) (...), ...)	Create a new Interrupt Context with its stack pointer set to $stackp + len$. Also create a new thread that can be swapped on to the CPU and return its identifier; this thread will begin execution in the function f. Used for creating new kernel threads, application threads, and processes.
sva.reinit.icontext (int (*f) (...), void * stackp, unsigned len)	Reinitialize an Interrupt Context so that it represents user-space state. On a return from interrupt, control will be transferred to the function f, and the stack pointer will be set to $stackp$.

1) No virtual addresses permitting write access can be mapped to frames containing native code translations.
2) The OS cannot create additional translations mapping virtual pages to native code frames.
3) Translations for virtual addresses used for the code segment cannot be modified.

Additional restrictions prevent the operating system from using the MMU to bypass the instrumentation on store instructions:

1) Translations for virtual addresses in KCoFI memory cannot be created, removed, or modified.
2) Translations involving the physical frames used to store data in KCoFI memory cannot be added, removed, or modified.

D. DMA and I/O Restrictions

Memory writes issued by the CPU are not the only memory writes that can corrupt the code segment or internal KCoFI memory. I/O writes to memory-mapped devices and external DMA devices can also modify memory. The KCoFI VM must control these memory accesses also.

The KCoFI design, like the original SVA design [5], uses an I/O MMU to prevent DMA operations from overwriting the OS kernel's code segment, the KCoFI memory, and frames that have been declared as page table pages.

Protecting KCoFI memory from I/O writes is identical to the instrumentation for memory writes; pointers are masked before dereference to ensure that they do not point into the KCoFI memory. Additionally, the KCoFI VM prevents reconfiguration of the I/O MMU. KCoFI instruments I/O port writes to prevent reconfiguration for I/O MMUs configured using I/O port writes; memory-mapped I/O MMUs are protected using the MMU. The KCoFI VM can therefore vet configuration changes to the I/O MMU like it does for the MMU.

E. Thread State

The KCoFI VM provides a minimal thread abstraction for representing the processor state. This structure is called a thread structure and is referenced by a unique identifier. Internally, as shown in Figure 3, a thread structure contains the state of the thread when it was last running on the CPU (including its program counter) and two stacks of Interrupt Contexts (described in Section IV-F).
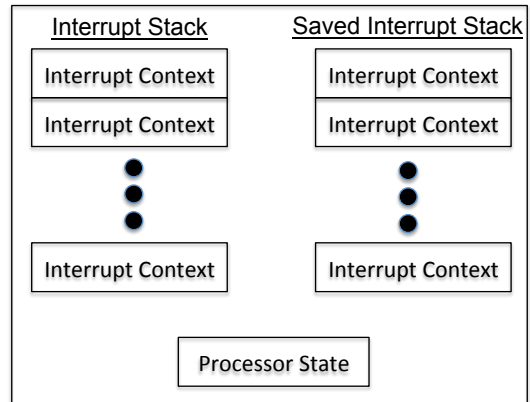

Fig. 3.   KCoFI Thread Structure

Thread structures are stored within the KCoFI memory to prevent direct modification by application or OS kernel code. The next few subsections will describe how the thread structure and the KCoFI instructions that manipulate them are used to provide interrupt, thread creation, and context switching operations that cannot violate control-flow integrity.

F. Protecting Interrupted Program State

When an interrupt, trap, or system call occurs, both the Linux and BSD operating systems store the interrupted program's state on the kernel stack [7], [8]. This state includes the return address at which execution should resume when the OS kernel decides to return from the interrupt, trap, or system call. Since it is stored in memory, this program counter value is vulnerable to corruption by memory safety errors.

Unlike other targets of indirect control transfers, the return address for a return-from-interrupt cannot be usefully determined via static analysis. Interrupts are allowed to occur at any time; any valid instruction location, in both application and kernel code, is permitted to be a valid return-from-interrupt target. The memory holding the return address must therefore be protected from corruption.

KCoFI saves the interrupted program state, called the *Interrupt Context*, on the Interrupt Context stack within the

currently active thread's thread structure within the KCoFI memory. KCoFI then switches the stack pointer to a predetermined kernel stack and transfers control to the OS kernel. Since the thread structure and stack of Interrupt Contexts are stored in KCoFI memory, the same bit-masking instrumentation used to protect the KCoFI memory is also used to protect the return address for interrupts, traps, and system calls.

OS kernels need to make controlled modifications to interrupted program state in order to dispatch signal handlers [7], [8], efficiently recover from page faults when copying data between user and kernel space [7], or restart interrupted system calls [7]. The SVA infrastructure provides instructions for making such controlled changes [5]; KCoFI provides new implementations of these instructions that do not rely on tracking memory object locations and sizes. These instructions are listed in Table II.

The `sva.ipush.function()` instruction modifies interrupted program state to push a function call frame on to the interrupted program's stack; it is used for signal handler dispatch. Our design, like the original SVA [5], only permits this modification to be made to an Interrupt Context representing user-space state.

Signal handler dispatch uses `sva.icontext.save()` and `sva.icontext.load()` to save and restore interrupted program state before and after signal handler dispatch. The Saved Interrupt Stack is used to save a copy of an original interrupt context before the original (on the Interrupt Stack) is modified to dispatch a signal. In particular, the `sva.icontext.save()` instruction makes a copy of the Interrupt Context at the top of the Interrupt Stack and pushes this copy on to the Saved Interrupt Stack within the thread structure. The `sva.icontext.load()` instruction will pop an Interrupt Context off the Saved Interrupt Context stack and replace the top-most element on the Interrupt Stack with this previously saved Interrupt Context, ensuring that the correct state is resumed on the next return from interrupt. Unlike `sva.icontext.save()`, we restrict `sva.icontext.load()` so that it can only load user-space interrupted program state back into the Interrupt Context (since signals in a commodity kernel are generally never dispatched to interrupted kernel code, only to userspace code).

Exception handling within the kernel is done using the LLVM `invoke` and `unwind` instructions. The `invoke` instruction is just a `call` instruction with an additional label to identify an exception handler. `invoke` transfers control flow to the called function; if that function (or one of its callees) throws an exception, it uses the `unwind` instruction to unwind control-flow on the stack to the most recently executed `invoke` instruction [20].

The `sva.iunwind` instruction can modify interrupted privileged state; it is equivalent to forcing the interrupted program to execute an `unwind` instruction. This behavior cannot cause control flow to deviate from the compiler's precomputed call graph and is therefore safe to use.

G. Thread Creation

When a commodity operating system creates threads, it performs two tasks that can affect control flow. First, it allocates and initializes a kernel stack and places data on the new stack to contain the state that, when restored on a return from system call, will start running the new user-space thread [7]. Second, it creates new kernel state that will be placed on to the processor on a context switch [7]; after the context switch, the new state will return from the system call, loading the new interrupted program state and returning back to the application.

KCoFI provides the `sva.init.icontext()` instruction for creating new threads and processes. This instruction first creates a new thread structure which can be swapped on to the CPU using the `sva.swap()` instruction discussed in Section IV-H. This native processor state within the new thread structure is initialized so that it will begin execution in the function passed to `sva.init.icontext()`; the supplied function pointer is checked to ensure that it points to the beginning of a function.

The `sva.init.icontext()` instruction also creates empty Interrupt and Saved Interrupt stacks within the new thread structure. It then creates a new Interrupt Context that is identical to the top-most Interrupt Context in the current thread's Interrupt Stack; it then pushes this new Interrupt Context on to the top of the Interrupt Stack in the new thread structure. This new Interrupt Context is then modified to use the stack specified in the call to `sva.init.icontext()`.

Finally, `sva.init.icontext()` verifies that the specified stack does not overlap with KCoFI memory. If the check passes, it initializes the new stack so that a return from the specified function will return into the KCoFI VM system call dispatching code. The configuration of the Interrupt Context will ensure that if the function returns that control-flow integrity is not violated. When the function returns, control flow will return to the KCoFI VM which will attempt to return from a system call, trap, or interrupt. If the new Interrupt Context was cloned from the initial Interrupt Context from the first thread executing at system boot, the Interrupt Context will have a program counter value of zero and will therefore fault, preventing a CFI violation. Otherwise, the new Interrupt Context will have a valid program counter value from the Interrupt Context from which it was duplicated, and therefore, the return from interrupt will succeed.

H. Context Switching

Context switching requires saving the current processor state into memory and loading new state on to the processor. The state, including the stack pointer and program counter, are vulnerable while residing in memory.

As Table III shows, KCoFI provides an instruction called `sva.swap()` that saves the current processor state into the thread structure within KCoFI memory and loads new state that has been saved by a previous call to `sva.swap()` or created by `sva.init.icontext()`. State is represented by opaque identifiers returned by the `sva.swap()` and `sva.init.icontext()` instructions. This prevents the `sva.swap()` instruction from loading invalid program state. By saving state inside the KCoFI VM memory, the program counter within the saved state cannot be corrupted by memory safety errors. The `sva.swap()` instruction disables interrupts

Name	Description
sva.swap (unsigned newID, unsigned * oldID)	Save the current processor native state and store an identifier representing it into *oldID* and then load the state represented by *newID*.

TABLE IV. KCoFI NATIVE CODE TRANSLATION INSTRUCTIONS

Name	Description
sva.translate (void * buffer, char * funcname, bool kmode)	Translate the SVA bitcode starting at buffer into native code. If *kmode* is true, then native code is generated for use in the processor's privileged mode. Otherwise, native code will be generated for use in the processor's unprivileged mode. The address to the function *funcname* will be returned.
sva.disable.privcode (void)	Disable further translation of SVA bitcode for use as native code in the processor's privileged state.

while it is executing, so that it cannot be interrupted and will never load inconsistent state.

The original SVA provides a similar instruction called `sva.swap.integer()` [5]. The primary difference between the SVA instruction and the KCoFI version is that KCoFI does not split the native processor state into individual components; it saves integer registers, floating point registers, and vector registers. While not necessary for control-flow integrity, it does ensure that the correct floating point and vector state is restored on context switching, providing applications with a more secure context switch.

I. Code Translation

Any OS code (e.g., the core kernel or a driver) to be loaded for execution must start out in SVA bitcode form, whereas a user program can be SVA bitcode or native code. When the OS needs to load and execute any piece of software, it first passes the code to the `sva.translate` intrinsic shown in Table IV. The intrinsic takes a Boolean argument indicating whether the code should be translated for use in user-space or kernel-space mode. If this flag is true, the intrinsic verifies that the code is in SVA bitcode form. If the code is SVA bitcode, `sva.translate` will translate the bitcode into native code and cache it offline for future use. `sva.translate` returns a pointer to the native code of function `funcname`.

If the function pointer points to kernel code, the kernel can call the function directly; this permits the use of dynamically loaded kernel modules. If the function pointer points to user-mode code, then the kernel must use the `sva.reinit.icontext()` instruction to set up a user-space Interrupt Context that will begin execution of the application code when the Interrupt Context is loaded on to the processor on the next return from interrupt. These mechanisms provide a way of securely implementing the `exec()` family of system calls.

While KCoFI already prevents traditional native code injection (because the KCoFI VM prevents bad control-transfers and disallows executable and writable memory), it must also prevent *virtual* code injection attacks. A virtual code injection attack uses a memory safety error to modify some SVA bitcode before it is passed to the `sva.translate()` intrinsic to trick the kernel into adding new, arbitrary code into the kernel.

To prevent such an attack, our design provides the `sva.disable.privcode()` instruction, which turns off

code generation for the kernel. This will allow the kernel to dynamically load bitcode files for drivers and extend its native code section during boot but prevent further driver loading after boot. A kernel that loads all of its drivers during boot would use this instruction immediately before executing the first user process to limit the time at which it would be vulnerable to virtual code injection attacks. (Note that the OS feature to hot-swap devices that require loading new device drivers might be precluded by this design.)

J. Installing Interrupt and System Call Handlers

Operating systems designate functions that should be called when interrupts, traps, and system calls occur. Like SVA [5], KCoFI provides instructions that allow the OS kernel to specify a function to handle a given interrupt, trap, or system call. These instructions first check that the specified address is within the kernel's virtual address space and has a CFI label. If the function address passes these checks, the instruction records the function address in a table that maps interrupt vector/system call numbers to interrupt/system call handling functions.

The hardware's interrupt vector table resides in KCoFI memory and directs interrupts into KCoFI's own interrupt and system call handling code. This code saves the interrupted program state as described in Section IV-F and then passes control to the function that the kernel designated.

V. FORMAL MODEL AND PROOFS

In order to demonstrate that key features of our design are correct, we built a model of the KCoFI virtual machine in the Coq proof assistant [22] and provide a partial proof that our design enforces control-flow integrity. The model and proofs comprise 2,008 non-comment lines of Coq code. Our proofs are checked mechanically by Coq.

As we are primarily interested in showing that our design in Section IV is correct, we model a simplified version of the KCoFI VM. While our model is simpler than and not proven sound with respect to the full implementation, it models key features for which formal reasoning about control-flow integrity has not previously been done; these features include virtual to physical address translation, trap entry and return, context switching, and signal handler dispatch.

TABLE V.   SUMMARY OF FORMAL MODEL SUPPORT FUNCTIONS

Function	Description
valid	$(v, pc, istack, sistack) \rightarrow v$
swapOn	$(v, n, istack, sistack) \times pc \rightarrow (true, pc, istack, sistack)$
swapOff	$(v, pc, istack, sistack) \rightarrow (false, 0, istack, sistack)$
ipush	$(v, pc, istack, sistack) \times ic \rightarrow (v, pc, ic :: istack, sistack)$
ipop	$(v, pc, ic :: istack, sistack) \rightarrow (v, pc, istack, sistack)$
itop	$(v, pc, ic :: istack, sistack) \rightarrow ic$
saveIC	$(v, pc, ic :: istack, sistack) \rightarrow (v, pc, ic :: istack, ic :: sistack)$
loadIC	$(v, pc, ic_1 :: istack, ic_2 :: sistack) \rightarrow (v, pc, ic_2 :: istack, sistack)$
getIPC	$(\mathcal{R}, pc) \rightarrow pc$
getIReg	$(\mathcal{R}, pc) \rightarrow \mathcal{R}$

Instructions   ::=   loadi n
                  | load n
                  | store n
                  | add n
                  | sub n
                  | map n tlb
                  | jmp
                  | jeq n
                  | jneg n
                  | trap
                  | iret
                  | svaSwap
                  | svaRegisterTrap
                  | svaInitIContext f
                  | svaSaveIContext
                  | svaLoadIContext
                  | svaPushFunction n

Fig. 4.   Instructions in KCoFI Model. Most instructions take the single register, $\mathcal{R}$, as an implicit operand.

In this section, we describe our model of the KCoFI virtual machine, our formal definition of control-flow integrity for operating system code, and our control-flow integrity proofs.

A. KCoFI Virtual Machine Model

Our machine model is a simplified version of the KCoFI virtual machine with the instruction set shown in Figure 4. To simplify the language and its semantics, we opted to use a simple assembly language instruction set for basic computation instead of the SSA-based SVA instruction set. Operations such as context switching and MMU configuration are performed using instructions similar to those described in Section IV. Our model does not include all the KCoFI features and does not model user-space code. However, it does include an MMU, traps, context switching, and the ability to modify and restore Interrupt Contexts as described in Section IV-F (which is used to support signal handler dispatch).

The physical hardware is modeled as a tuple called the *configuration* that represents the current machine state. The configuration contains:

- the value of a single hardware register $\mathcal{R}$

- a program counter $\mathcal{PC}$

- a memory (or store) σ that maps physical addresses to values

- a software TLB μ that maps virtual addresses to TLB entries. A TLB entry is a tuple containing a physical address and three booleans that represent read, write, and execute permission to the physical address. The function ρ returns the physical address within a TLB entry while the functions RD(), WR(), and EX() return true if the TLB entry permits read, write, and execute access, respectively. Unlike real hardware, our model's MMU maps virtual to physical addresses at byte granularity.

- a set of virtual addresses $\mathcal{CFG}$ to which branches and traps may transfer control flow. All new threads must begin execution at a virtual address within $\mathcal{CFG}$.

- a pair (cs, ce) marking the first and last physical address of the kernel's code segment

- a current thread identifier $\mathcal{T}$

- a new thread identifier $\mathcal{NT}$

- a function τ that maps a thread identifier to a thread. A thread is a tuple $(v, pc, istack, sistack)$ in which v is a boolean that indicates whether a thread can be context switched on to the CPU and pc is the program counter at which execution should resume when the thread is loaded on to the CPU. The $istack$ is the Interrupt Context stack in Figure 3 used when traps and returns from traps occur. The $sistack$ is the Saved Interrupt Stack in Figure 3 and stores Interrupt Contexts that are saved by svaSaveIContext.

- a virtual address $\mathcal{TH}$ that is the address of the trap handler function

Since the configuration is large, we will replace one or more elements with an ellipsis (i.e., ...) as necessary to keep the text concise.

An Interrupt Context is a tuple that represents a subset of the configuration. It contains a copy of the machine's single register and the program counter. Interrupt Contexts are stored in stacks with standard push/pop semantics. The special value nil represents an empty stack; attempting to pop or read the top value of an empty stack results in an Interrupt Context with zero values.

There are several support functions, summarized in Table V, that help make our semantics easier to read. The *valid* function takes a thread $\mathcal{T}$ and returns the value of its boolean flag. The *swapOn* function takes a thread $\mathcal{T}$ and an integer and

returns an identical thread with its boolean flag set to true and its program counter set to the specified integer. Conversely, the *swapOff* function takes a thread and returns a thread that is identical except for its boolean being set to false and its program counter being set to zero. The *ipush* function takes a thread and an Interrupt Context and returns a thread with the Interrupt Context pushed on to the $istack$ member of the thread tuple. The *ipop* function is similar but pops the top-most Interrupt Context off the thread's $istack$. The *saveIC* function takes a thread and returns an identical thread in which the top-most element of the $istack$ member is pushed on to the $sistack$ member. The *loadIC* function pops the top-most Interrupt Context from the $sistack$ member and uses that value to replace the top-most member of the $istack$ member. The *itop* function takes a thread and returns the top-most Interrupt Context on its $istack$. Finally, the *getIPC* and *getIReg* functions take an Interrupt Context and return the program counter and register value stored within the Interrupt Context, respectively.

One feature of our configuration is that the KCoFI VM's internal data structures are not stored in memory; they are instead part of the configuration and can therefore not be modified by the `store` instruction. An advantage of this approach is that our proofs demonstrate that CFI is enforced regardless of the mechanism employed to protect these data structures (i.e., it shows that if these data structures are protected, then the proof holds). The disadvantage of this approach is that it does not prove that our sandoxing instrumentation on stores is designed correctly. However, given the simplicity of our instrumentation, we believe that having a simpler, more general proof is the better tradeoff.

B. Instruction Set and Semantics

The instruction set is shown in Figure 4. The `loadi` instruction loads the constant specified as its argument into the register; the `load` instruction loads the value in the virtual address given as its argument into the register. The `store` instruction stores the value in the register to the specified virtual memory address. The `add` (`sub`) instruction adds (subtracts) a constant with the contents of the register and stores the result in the register. The `map` instruction modifies a virtual to physical address mapping in the software-managed TLB. The `jmp` instruction unconditionally transfers control to the address in the register while the `jeq` and `jneg` instructions transfer control to the specified address if the register is equal to zero or negative, respectively.

A subset of the KCoFI instructions are also included in the instruction set. Some of the instructions differ from their KCoFI implementations because our formal model does not have an implicit stack whereas the KCoFI instruction set does. Our model also has `trap` and `iret` instructions for generating traps and returning from trap handlers.

The semantic rules for each instruction are specified as a state transition system. The transition relation $c_1 \Rightarrow c_2$ denotes that the execution of an instruction can move the state of the system from configuration c_1 to configuration c_2.

Figure 5 shows the semantic rules for each instruction. Each rule has a brief name describing its purpose, the con-

ditions under which the rule can be used, and then the actual transition relation.

Each rule essentially fetches an instruction at the address of the program counter, checks for safety conditions (given as part of the premise of the implication), and then generates a new state for the machine to reflect the behavior of the instruction.

All instructions require that the program counter point to a virtual address with execute permission. Loads and stores to memory require read or write access, respectively. The jump instructions always check that the destination is a valid target. The map instruction is allowed to change a virtual to physical page mapping if the virtual address given as an argument is not already mapped to a location within the code segment *and* it does not permit a new virtual address to map to an address within the code segment.

C. Control-Flow Integrity Theorems

We now outline our control-flow integrity proofs for this system. Our first two proofs ensure that each transition in the semantics (i.e., the execution of a *single* instruction) maintains control-flow integrity.

There are several invariants that must hold on a configuration if the transition relation is to maintain control-flow integrity. For example, the system must not start in a state with a writeable code segment. We therefore define five invariants that should hold over all configurations:

Invariant 1. *VC(c): For configuration c = (..., cs, ce, ...), $0 < cs \leq ce$.*

Invariant 2. *TNW(c): For configuration c = (μ, σ, Reg, ..., cs, ce, ...), $\forall n : cs \leq \rho(\mu(n)) \leq ce, \neg WR(\mu(n))$*

Invariant 3. *TMAP1(c): For configuration c = (μ, σ, ..., cs, ce, ...), $\forall n\, m : cs \leq \rho(\mu(n)) \leq ce \land n \neq m, \rho(\mu(n)) \neq \rho(\mu(m))$*

Invariant 4. *TH(c): For configuration c = (..., $\mathcal{TH}$), $\mathcal{TH} \in CFG$*

Invariant 5. *THR(c): For configuration c = (μ, σ, ..., CFG, ..., τ, ...), $\forall$ (v, pc, istack, sistack) $\in \tau$: pc $\in CFG \lor \sigma(\rho(\mu (pc - 1))) = svaSwap$*

Invariant 1 states that the start of the code segment must be non-zero and less than or equal to the end of the code segment. Invariant 2 asserts that there are no virtual-to-physical address mappings that permit the code segment to be written. Invariant 3 asserts that there is at most one virtual address that is mapped to each physical address within the code segment. Invariant 4 ensures that the system's trap handler is an address that can be targeted by a branch instruction.

Invariant 5 restricts the value of the program counter in saved thread structures. A newly created thread needs to have an address at which to start execution; Invariant 5 restricts this value to being an address within CFG. A thread that has been swapped off the CPU should have a program counter that points to the address immediately following the `svaSwap` instruction. The second half of the disjunction in Invariant 5 permits this.

Control-flow integrity in our system covers two key properties. First, each instruction should transfer control to one of

LoadImm: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ loadi n $\rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, n, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

Load: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ load n $\wedge RD(\mu(n)) \wedge \sigma(\rho(\mu(n))) = $ val v $\rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, v, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

Store: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ store n $\wedge WR(\mu(n)) \rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma[\rho(\mu(n)) \leftarrow (val \; \mathcal{R})], \mathcal{R}, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

Add: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ add n $\rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R} + n, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

Sub: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ sub n $\rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R} - n, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

Jump: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ jmp $\wedge \mathcal{R} \in CFG \rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R}, \mathcal{R}, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

JumpEq1: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ jeq v $\wedge v \in CFG \rightarrow$
$(\mu, \sigma, 0, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, 0, v, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

JumpEq2: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ jeq v $\wedge \mathcal{R} \neq 0 \rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R}, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

JumpNeg1: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ jneg v $\wedge v \in CFG \wedge \mathcal{R} < 0 \rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R}, v, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

JumpNeg2: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ jneg v $\wedge \mathcal{R} \geq 0 \rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R}, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

Map: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ map v tlb $\wedge \neg (cs \leq \rho(tlb) \leq ce) \wedge \neg (cs \leq \rho(\mu(v)) \leq ce) \rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu[v \leftarrow tlb], \sigma, \mathcal{R}, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH})$

Swap: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ svaSwap $\wedge$ valid$(\tau(\mathcal{R})) \rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{T}, getPC(\tau(\mathcal{R})), CFG, cs, ce, \mathcal{R}, \mathcal{NT}, \tau[\mathcal{T} \leftarrow swapOn(\tau(\mathcal{T}), PC + 1)] [\mathcal{R} \leftarrow swapOff(\tau(\mathcal{R}))], \mathcal{TH})$

Trap: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ trap $\rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R}, \mathcal{TH}, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau[\mathcal{T} \leftarrow ipush(\tau(\mathcal{T}),(\mathcal{R},PC + 1))], \mathcal{TH})$

IRet: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ iret $\rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, getIReg(itop(\tau(\mathcal{T}))), getIPC(itop(\tau(\mathcal{T}))), CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau[\mathcal{T} \leftarrow ipop(\tau(\mathcal{T}))], \mathcal{TH})$

RegisterTrap: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ svaRegisterTrap $\wedge \mathcal{R} \in CFG \rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R}, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{R})$

InitIContext: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ svaInitIContext f $\wedge f \in CFG \rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{NT}, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT} + 1, \tau[\mathcal{NT} \leftarrow (true, f, itop(\tau(\mathcal{T})) :: nil, nil)], \mathcal{TH})$

SaveIContext: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ svaSaveIContext $\rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R}, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau[\mathcal{T} \leftarrow saveIC(\tau(\mathcal{T}))], \mathcal{TH})$

LoadIContext: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ svaLoadIContext $\rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R}, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau[\mathcal{T} \leftarrow loadIC(\tau(\mathcal{T}))], \mathcal{TH})$

PushIContext: $EX(\mu(PC)) \wedge \sigma(\rho(\mu(PC))) = $ svaPushFunction a $\rightarrow$
$(\mu, \sigma, \mathcal{R}, PC, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau, \mathcal{TH}) \Rightarrow (\mu, \sigma, \mathcal{R}, PC + 1, CFG, cs, ce, \mathcal{T}, \mathcal{NT}, \tau[\mathcal{T} \leftarrow ipush(\tau(\mathcal{T}), (a, \mathcal{R}))], \mathcal{TH})$

Fig. 5.   KCoFI Instruction Semantics

four locations: the virtual address of the subsequent instruction, a valid target within the pre-computed control-flow graph, an instruction following an `svaSwap` instruction, or the program counter value stored in the top-most interrupt context of the current thread (which cannot be corrupted since it does not reside in memory). Theorem 1 states this more formally:

Theorem 1. $\forall \; c_1 = (\mu_1, \sigma_1, ..., PC_1, ..., CFG, ..., \mathcal{T}_1, ..., \tau_1)$, $c_2 = (\mu_2, \sigma_2, ..., PC_2, ..., CFG, ..., \mathcal{T}_2, ..., \tau_2) : (c_1 \Rightarrow c_2) \wedge TH(c_1) \wedge THR(c_1) \rightarrow PC_2 = PC_1 + 1 \vee PC_2 \in CFG \vee \sigma_2(\rho(\mu_2(PC_2 - 1))) = svaSwap \vee PC_2 = getIPC(itop(\tau_1(\mathcal{T}_1)))$

Second, we want to ensure that the instruction stream read from the code segment does not change due to writes by the store instruction or by reconfiguration of the MMU. In other words, we want to ensure that reading from a virtual address that maps into the code segment reads the same value after executing an instruction as it held before execution of the instruction. Theorem 2 states this formally as:

Theorem 2. $\forall \; v, \; c_1 = (\mu_1, \sigma_1, ..., cs, ce, ...), \; c_2 = (\mu_2, \sigma_2, ...) : (c_1 \Rightarrow c_2) \wedge VC(c_1) \wedge TNW(c_1) \wedge TMAP1(c_1) \wedge cs \leq \rho(\mu_1(v)) \leq ce \rightarrow \sigma_1(\rho(\mu_1(v))) = \sigma_2(\rho(\mu_2(v)))$

We proved that both Theorem 1 and 2 hold true in the Coq proof assistant. Intuitively, Theorem 1 is true because all the jump instructions check the target address against the precomputed control-flow graph while the `svaSwap` instruction always saves and loads the correct program value from saved processor state. The intuition behind Theorem 2 is that the checks on the `map` instruction prevent a) addresses that are already mapped into code segment from being remapped to different addresses, and b) new virtual-to-physical address mapping from making parts of the code segment writable.

While proving Theorems 1 and 2 shows that the restrictions on the instructions maintain control-flow integrity for single instruction executions, a full control-flow integrity proof will demonstrate that control-flow integrity is maintained on the transitive closure of the transition relation (in other words, if the system starts in a state satisfying the required invariants, then control-flow integrity is maintained after executing an arbitrary number of instructions). We therefore also need to prove that the transition relation preserves the invariants. We have therefore proven the following additional theorems:

Theorem 3. $\forall \; c_1, \; c_2: (c_1 \Rightarrow c_2) \wedge VC(c_1) \rightarrow VC(c_2)$

Theorem 4. $\forall \; c_1, \; c_2: (c_1 \Rightarrow c_2) \wedge TNW(c_1) \rightarrow TNW(c_2)$

Theorem 5. $\forall \; c_1, \; c_2: (c_1 \Rightarrow c_2) \wedge TMAP1(c_1) \rightarrow TMAP1(c_2)$

Theorem 6. $\forall \; c_1, \; c_2: (c_1 \Rightarrow c_2) \wedge TH(c_1) \rightarrow TH(c_2)$

Proving that Invariant 5 holds across the transition relation requires additional invariants to hold on the configuration. These new invariants are:

Invariant 6. *CFGT(c): For configuration* $c = (\mu, ..., CFG, cs, ce, ...)$, $\forall \; v: v \in CFG \rightarrow cs \leq \rho(\mu(v)) \leq ce$.

Invariant 7. *PCT(c): For configuration* $c = (\mu, ..., PC, ..., cs, ce, ...)$, $cs < \rho(\mu(PC)) \leq ce$.

Invariant 8. *tlText(c): For configuration* $c = (\mu, ..., CFG, cs, ce, ..., \tau, ...)$, $\forall \; (v, pc, istack, sistack) \in \tau: cs \leq \rho(\mu(pc)) \leq ce \wedge ((pc \in CFG) \vee cs \leq \rho(\mu(pc - 1)) \leq ce)$

Invariants 6 and 7 state that the list of valid indirect branch targets and the machine's program counter are all virtual addresses that are mapped to the code segment. Invariant 8 states that all swapped-off threads also have program counters that are within the code segment; it also ensures that threads with program counters that are not within the valid list of indirect branch targets have their previous program counter within the code segment (i.e., the `svaSwap` instruction that swapped the thread off the CPU is also in the code segment).

Invariants 6, 7, and 8 suffice for showing that each swapped-off thread has a valid program counter value. We have therefore formally proven using Coq that Invariant 5 holds across all instruction executions:

Theorem 7. $\forall \; c_1, \; c_2: (c_1 \Rightarrow c_2) \wedge VC(c_1) \wedge TNW(c_1) \wedge TMAP1(c_1) \wedge PCT(c_1) \wedge CFGT(c_1) \wedge tlText(c_1) \wedge THR(c_1) \rightarrow THR(c_2)$

We have proved using Coq that Invariant 6 holds across the transition relation:

Theorem 8. $\forall \; c_1, \; c_2: (c_1 \Rightarrow c_2) \wedge CFGT(c_1) \rightarrow CFGT(c_2)$

Proving that Invariants 7 and 8 hold across the transition relation and completing the proof that control-flow integrity is maintained across the transitive closure of the transition relation is left to future work. However, Theorems 3, 4, and 5 permit us to prove that the code segment is not modified over the reflexive and transitive closure of the transition relation, denoted $\Rightarrow^*$. We have therefore proven the following theorem using Coq:

Theorem 9. $\forall \; v, \; c_1 = (\mu_1, \sigma_1, ..., cs, ce, ...), \; c_2 = (\mu_2, \sigma_2, ...) : (c_1 \Rightarrow^* c_2) \wedge VC(c_1) \wedge TNW(c_1) \wedge TMAP1(c_1) \wedge cs \leq \rho(\mu_1(v)) \leq ce \rightarrow \sigma_1(\rho(\mu_1(v))) = \sigma_2(\rho(\mu_2(v)))$

VI. IMPLEMENTATION

We implemented a new version of the SVA-OS instructions and run-time for 64-bit x86 processors. The implementation runs 64-bit code only. This new implementation reuses code from the original 32-bit, single-core implementation [12], [5]. This new implementation only supports a single CPU system at present, but that is mainly due to needing to find a good locking discipline for the MMU instructions; all the other features maintain per-CPU data structures to permit multi-processor and multi-core functionality.

We ported FreeBSD 9.0 to the SVA-OS instruction set. We chose FreeBSD over Linux because the Clang/LLVM compiler can compile an unmodified FreeBSD kernel.

We used the sloccount tool [23] from the FreeBSD ports tree to measure the size of our TCB. Excluding comments and white space, our system contains 4,585 source lines of code for the KCoFI run-time library linked into the kernel and an additional 994 source lines of code added to the LLVM compiler to implement the compiler instrumentation. In total, our TCB is 5,579 source lines of code.

A. Instrumentation

The CFI and store instrumentation is implemented in two separate LLVM passes. The CFI instrumentation pass

is a version of the pass written by Zeng et. al. [18] that we updated to work on x86_64 code with LLVM 3.1. The store instrumentation pass is an LLVM IR level pass that instruments store and atomic instructions that modify memory; it also instruments calls to LLVM intrinsic functions such as `llvm.memcpy`.

We modified the Clang/LLVM 3.1 compiler to utilize these instrumentation passes when compiling kernel code. To avoid the need for whole-program analysis, we use a very conservative call graph: we use one label for all call sites (i.e., the targets of returns) and for the first address of every function. While conservative, this callgraph allows us to measure the performance overheads and should be sufficient for stopping advanced control-data attacks.

Unlike previous work [18], [9], we use the sequence `xchg %rcx, %rcx ; xchg %rdx, %rdx` to create a 32-bit label. We found that this sequence is both easy to implement (since they are NOPs, these instructions do not overwrite any register values) and much faster than a 64-bit version of the `prefetchnta` sequence used in previous work [18].

B. KCoFI Instruction Implementation

The KCoFI instructions described in Section IV are implemented in a run-time library that is linked into the kernel at compile-time. The semantics for the instructions given in Section V assume that the KCoFI instructions execute atomically. For that reason, the run-time library implementations disable hardware interrupts when they start execution and re-enable them upon exit as the original SVA implementation did [5].

C. Interrupt Context Implementation

To place the Interrupt Context within the KCoFI VM internal memory, we use the Interrupt Stack Table (IST) feature of the x86_64 [24], as was done in Virtual Ghost [25]. This feature causes the processor to change the stack pointer to a specific location on traps or interrupts regardless of whether a processor privilege mode change has occurred. The KCoFI VM uses this feature to force the processor to save state within KCoFI's internal memory before switching to the real kernel stack.

Unlike previous versions of SVA [12], [5], KCoFI saves all native processor state on every interrupt, trap, and system call. This includes the x86_64 general purpose registers, the x87 FPU registers, and the XMM and SSE registers. We believe an improved implementation can save the floating point and vector state lazily as the native FreeBSD 9.0 kernel does, but that improvement is left to future work.

D. Unimplemented Features

Our implementation does not yet include the protections needed for DMA. However, we believe that I/O MMU configuration is rare, and therefore, the extra protections for DMA should not add noticeable overhead. Our implementation also lacks the ability to translate SVA bitcode (or to look up cached translations for bitcode) at run-time. Instead, our current implementation translates all OS components to native code ahead-of-time, and runs only native-code applications.

For ease of implementation, we add the same CFI labels to both kernel code and the SVA-OS run-time library. While this deviates from the design, it does not change the performance overheads and makes the security results more conservative (no better and possibly worse).

VII. SECURITY EVALUATION

We performed two empirical evaluations to measure the security of our approach. Since KCoFI does not permit memory to be both readable and executable, we focus on return-oriented programming attacks.

Our first evaluation examines how well KCoFI removes instructions from the set of instructions that could be used in a return-oriented programming attack (which can work with or without return instructions [26]). We compute a metric that summarizes this reduction called the average indirect target reduction (AIR) metric [10].

Since not all instructions are equally valuable to an attacker, we performed a second evaluation that finds instruction sequences (called *gadgets* [3]) that can be used in an ROP attack and determines whether they can still be used after KCoFI has applied its instrumentation.

A. Average Indirect Target Reduction

Return oriented programming attacks work because of the plethora of instructions available within a program. To get a sense of how many instructions we removed from an attacker's set of usable instructions, we used Zhang and Sekar's AIR metric [10]; this metric computes the average number of machine code instructions that are eliminated as possible targets of indirect control transfers. The AIR metric quantifies the reduction in possible attack opportunities in a way that is independent of the specific CFI method employed, the compiler used, and the architecture.

Equation 1 from Zhang and Sekar [10] provides the general form for computing the AIR metric for a program. n is the number of indirect branch instructions in the program, S is the total number of instructions to which an indirect branch can direct control flow before instrumentation, and $|T_i|$ is the number of instructions to which indirect branch i can direct control flow after instrumentation:

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

Since all indirect branch instructions instrumented by KCoFI can direct control-flow to the same set of addresses, Equation 1 can be simplified into Equation 2 (with $|T|$ being the number of valid targets for each indirect branch):

$$1 - \frac{|T|}{S} \qquad (2)$$

We measured the AIR metric for the KCoFI native code generated for the FreeBSD kernel. Our compiler identified 106,215 valid native code targets for indirect control flow transfers ($|T|$) out of 5,838,904 possible targets in the kernel's code segment (S) before instrumentation. The native code generated by KCoFI contains 21,635 indirect control flow transfers (n). The average reduction of targets (AIR metric) for

these transfers is therefore 98.18%, implying that nearly all the possible indirect control transfer targets have been eliminated as feasible targets by KCoFI.

As a point of comparison, our AIR result is nearly as good as the average AIR metrics for several different CFI variants reported for the SPEC CPU 2006 benchmarks and the namd benchmark (which range between 96% to 99.1%) [10]. Since these numbers are obtained for very different workloads – SPEC and the FreeBSD kernel – the comparison is only intended to show that the results are roughly similar; the differences in the exact numbers are not meaningful.

B. ROP Gadgets

To measure the impact on return-oriented-programming opportunities more specifically, we used the open-source ROP-Gadget tool [14] version 4.0.4 to automatically find ROP gadgets in both the original FreeBSD kernel compiled with GCC and our identically configured KCoFI FreeBSD kernel. We ran the tool on both the kernel and drivers using the default command-line options.

ROPGadget found 48 gadgets in the original FreeBSD kernel and 21 gadgets in the KCoFI FreeBSD kernel. We manually analyzed the 21 gadgets found in the KCoFI FreeBSD kernel. None of the gadgets follow a valid control-flow integrity label. Therefore, none of these gadgets can be "jumped to" via an indirect control transfer in an ROP attack.

VIII. PERFORMANCE EVALUATION

We evaluated the performance impact of KCoFI on a Dell Precision T1650 workstation with an Intel® Core™ i7-3770 processor at 3.4 GHz with 8 MB of cache, 16 GB of RAM, an integrated PCIE Gigabit Ethernet card, a 7200 RPM 6 Gb/s SATA hard drive, and a Solid State Drive (SSD) used for the /usr partition. For experiments requiring a network client, we used an iMac with a 4-core hyper-threaded Intel® Core™ i7 processor at 2.6 GHz with 8 GB of RAM. Our network experiments used a dedicated Gigabit ethernet network.

Since network applications make heavy use of operating system services, we measured the performance of the thttpd web server and the remote secure login sshd server. These experiments also allow us to compare the performance of KCoFI to the original SVA system [5] which enforces more sophisticated memory safety that implies control-flow integrity.

To measure file system performance, we used the Postmark benchmark [27]. We used the LMBench microbenchmarks [28] to measure the performance of individual system calls.

For each experiment, we booted the Dell machine into single-user mode to avoid having other system processes affect the performance of the system. Our baseline is a native FreeBSD kernel compiled with the LLVM 3.1 compiler, with the same compiler options and the same kernel options as the KCoFI FreeBSD kernel.

A. Web Server Performance

We used a statically linked version of the thttpd web server [29] to measure how much the KCoFI run-time checks

reduce the server's bandwidth. To measure bandwidth, we used ApacheBench [30].

For the experiments, we transferred files between 1 KB and 2 MB in size. Using larger file sizes is not useful because the network saturates at about 512KB file sizes. This range of sizes also subsumes the range used in the original SVA experiments [5]. We generated each file by collecting random data from the /dev/random device; this ensures that the file system cannot optimize away disk reads due to the file having blocks containing all zeros. We configured each ApacheBench client to make 32 simultaneous connections and to perform 10,000 requests for the file; we ran four such ApacheBench processes in parallel for each run of the experiment to simulate multiple clients. We ran each experiment 20 times.

Figure 6 shows the mean performance of transferring a file of each size. The average bandwidth reduction across all file sizes is essentially zero. This is far better performance than the SVA system which incurs about a 25% reduction in bandwidth due to its memory safety checks [5].
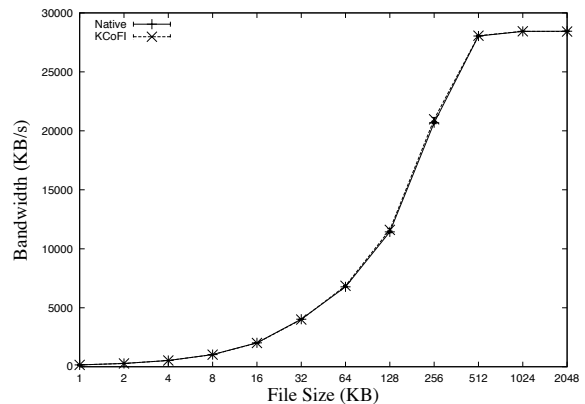

Fig. 6. ApacheBench Average Bandwidth with Standard Deviation Bars

B. Secure Shell Server Performance

In addition to a web server, we also measured the bandwidth of transferring files using the OpenSSH Secure Shell server [31]. We ran the OpenSSH server on our test machine and used the Mac OS X OpenSSH scp client (based on OpenSSH 5.2p1) to measure the number of bytes received per second when transferring the file. We repeated each experiment 20 times.

Figure 7 plots the mean bandwidth for the baseline system and KCoFI with standard deviation error bars (the standard deviations are too small to be discernible in the diagram). On average, the bandwidth reduction was 13% with a worst case reduction of 27%. Transferring files between 1 KB and 8 KB showed the most overhead at 27%. Transferring files that are 1 MB or smaller showed an average overhead of 23%; the average is 2% for files of larger size, indicating that the network becomes the bottleneck for larger file transfers.

The original SVA system only measured SSH bandwidth for files that were 8 MB or larger [5]; this is beyond the point at which the network hardware becomes the bottleneck. This comparison, therefore, is inconclusive: it does not show any difference between the two systems, but it does not include cases where overheads might be expected.
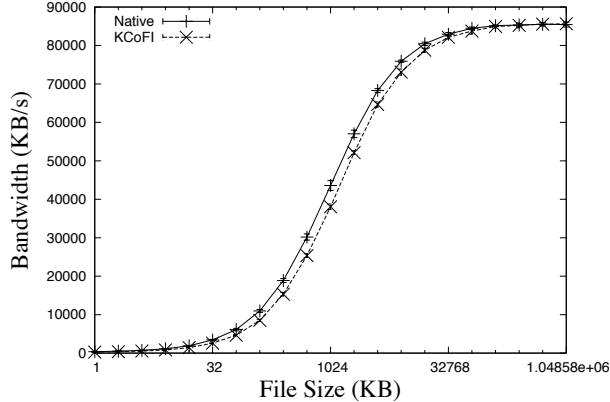
Fig. 7. SSHD Average Transfer Rate with Standard Deviation Bars

C. Microbenchmarks

In order to understand how our system affects the performance of core operating system services, we used LM-Bench [28] to measure the latency of various system calls. (We present these before discussing Postmark because the latter is largely explained by the LMBench measurements.) Some test programs can be configured to run the test for a specified number of iterations; those were configured to use 1,000 iterations. We ran each benchmark 10 times. We configured file I/O benchmarks to use files on the SSD. This ensures that we're measuring the highest relative latency that KCoFI can add by using the fastest disk hardware available.

TABLE VI.   LMBench Results. Time in Microseconds.

Test	Native	KCoFI	Overhead	SVA Overhead [5]
null syscall	0.091	0.22	2.50x	2.31x
open/close	2.01	4.96	2.47x	11.0x
mmap	7.11	23.4	3.30x	-
page fault	31.6	35.2	1.11x	-
signal handler install	0.168	0.36	2.13x	5.74x
signal handler delivery	1.27	1.17	0.92x	5.34x
fork + exit	62.9	222	3.50x	-
fork + exec	101	318	3.10x	-
select	3.05	4.76	1.60x	8.81x
pipe latency	1.94	4.01	2.10x	13.10x

TABLE VII.   LMBench: Files Creations Per Second

File Size	Native	KCoFI	Overhead
0 KB	155771	68415	2.28x
1 KB	97943	39615	2.47x
4 KB	97192	39135	2.48x
10 KB	85600	35982	2.38x

As Tables VI and VII show, our system can add considerable overhead to individual operations. Most of the operations we tested showed overhead between 2x and 3.5x. The KCoFI file creation overheads are uniformly about 2.3-2.5x for all file sizes tested by LMbench. Although these overheads are fairly high, most applications only experience these overheads during kernel CPU execution, which explains why the impact on performance observed for `thttpd` and `sshd` is far lower.

We also compared these results to similar benchmarks from the full memory-safety version of SVA [5], shown in the last column of Table VI. The missing numbers for SVA are because some kernel operations were not tested in the SVA experiments. On these microbenchmarks, KCoFI clearly performs much better than SVA, as much as 5x in some cases. Again, the SVA experiments used a different kernel and so it is not meaningful to compare the detailed differences in the numbers, but the magnitudes of the differences clearly highlight the performance benefit of using CFI instead of full memory safety.

D. Postmark Performance

To further examine file system performance, we ran Postmark [27] which mimics a mail server's file system behavior.

TABLE VIII.   Postmark Results

Native (s)	Native StdDev	KCoFI (s)	KCoFI StdDev	Overhead
12.7	0.48	24.8	0.40	1.96x

We configured Postmark to use 500 base files with sizes ranging from 500 bytes to 9.77 KB with 512 byte block sizes. The read/append and create/delete biases were set to 5, and we configured Postmark to use buffered file I/O. We ran Postmark on the SSD since it has lower latency and less variability in its latency than the hard disk. Each run of the experiment performed 500,000 transactions. We ran the experiment 20 times on both the native FreeBSD kernel and the KCoFI system. Table VIII shows the average results.

As Table VIII shows, the Postmark overheads are close to the LMBench file creation overheads.

IX. Related Work

Abadi et. al. [4] introduced the definition of control-flow integrity and proved that their approach enforced context-insensitive control-flow integrity. Our proof for the KCoFI design uses a similar approach but also demonstrates how control-flow integrity is preserved during OS operations that can have complex, often unanalyzable, impact on control flow, including context switching, MMU configuration, signal handler dispatch, and interrupts.

Zhang and Sekar's BinCFI [10] and Zhang et. al.'s CC-FIR [32] transform binary programs to enforce CFI. Similarly, Strato [9] modifies the LLVM compiler to instrument code with CFI checks similar to those used by KCoFI. None of these techniques can protect against ret2usr attacks since they find the targets of control-flow transfers via static analysis. KCoFI does not verify that its instrumentation is correct like Strato does [9]. However, KCoFI can incorporate Strato's instrumentation verification techniques.

RockSalt [33] is a verified verifier for Google's Native Client [34]; the verifier ensures that Native Client x86 machine code enforces sandboxing and control-flow integrity properties and has been proven correct via formal proof. Native Client's CFI policy [34] only requires that the x86 segment registers be left unmodified and that branches jump to aligned instructions; its CFI policy is therefore less restrictive than KCoFI's policy,

and it does not permit code to perform the myriad of operations that an OS kernel must be able to perform.

The Secure Virtual Architecture [12], [5] provides strong control-flow integrity guarantees. However, it also enforces very strong memory safety properties and sound points-to analysis; this required the use of whole-program pointer analysis [13] which is challenging to implement. SafeDrive [35] also enforces memory safety on commodity OS kernel code. However, SafeDrive requires the programmer to insert annotations indicating where memory object bounds information can be located. These annotations must be updated as the code is modified or extended.

HyperSafe [6] enforces control-flow integrity on a hypervisor. Like SVA [5], HyperSafe vets MMU translations to protect the code segment and interrupt handler tables; it also introduces a new method of performing indirect function call checks. HyperSafe, however, does not protect the return address in interrupted program state, so it does not fully implement CFI guarantees and does not prevent ret2usr attacks. Furthermore, HyperSafe only protects a hypervisor, which lacks features such as signal handler delivery; KCoFI protects an entire commodity operating system kernel.

kGuard [11] prevents ret2usr attacks by instrumenting kernel code to ensure that indirect control flow transfers move control flow to a kernel virtual address; it also uses diversification to prevent attacks from bypassing its protection and to frustrate control-flow hijack attacks. KCoFI uses similar bit-masking as kGuard to prevent user-space native code from forging kernel CFI labels. kGuard also uses diversification to prevent their instrumentation from being bypassed, which provides probabilistic protection against ROP attacks (with relatively low overhead), whereas KCoFI provides a CFI guaranty and ensures that the kernel's code segment is not modified.

Giuffrida et. al. [36] built a system that uses fine-grained randomization of the kernel code to protect against memory safety errors. Their system's security guarantees are probabilistic while our system's security guarantees are always guaranteed. Additionally, their prototype has only been applied to Minix while ours has been applied to a heavily used, real-world operating system (FreeBSD).

SecVisor [37] prevents unauthorized code from executing in kernel space but does not protect loaded code once it is executing. Returnless kernels [38] modify the compiler used to compile the OS kernel so that the kernel's binary code does not contain return instructions. Such kernels may still have gadgets that do not utilize return instructions [26].

The seL4 [39] microkernel is written in a subset of C and both the design and implementation are proven functionally correct, using an automated proof assistant. The proof ensures that the code does not have memory safety errors that alter functionality [39]. While seL4 provides stronger security guarantees than KCoFI, it only provides them to the microkernel while KCoFI provides its guarantees to a commodity OS kernel of any size. Changes to the seL4 code must be correct and require manual updates to the correctness proof [39] while KCoFI can automatically reapply instrumentation after kernel changes to protect buggy OS kernel code.

Several operating systems provide control-flow integrity by virtue of being written in a memory-safe programming language [40], [41], [42], [43], [44]. Verve [44], the most recent, is a kernel written in a C#-like language that sits upon a hardware abstraction layer that has been verified to maintain the heap properties needed for memory safety. While KCoFI can enforce control-flow integrity, its implementation is not verified like Verve's hardware abstraction layer.

X. Future Work

There are several plans for future work. First, we plan to investigate improvements to KCoFI's speed and efficacy. For example, using separate stacks for control-data and local variables could both improve performance and enforce a more restrictive context-sensitive CFI policy. We also plan to do much more low-level tuning of the system's performance than we have done; there is room for extensive improvement.

Second, we plan to finish the control flow integrity proof in Section V so that it proves that control flow integrity is maintained across the transitive closure of the transition relation. Furthermore, we plan to enhance the formal model to include more features (such as user-space application support).

Third, we plan to investigate building a verified *implementation* of KCoFI. Similar work has been done with operating systems written in safe languages [44]; while an ambitious goal, doing the same for existing commodity operating systems could help uncover implementation bugs and would increase confidence in the system's security.

XI. Conclusions

In this paper, we presented KCoFI: a system which provides comprehensive control flow integrity to commodity operating systems. We have shown that KCoFI provides protection to OS kernel code similar to that found for user-space code with better overheads than previously developed techniques for commodity OS kernels. Essentially, KCoFI uses traditional label-based protection for programmed indirect jumps but adds a thin run-time layer linked into the OS that protects key OS kernel data like interrupted program state and monitors all low-level state manipulations performed by the OS. We have provided a partial proof that KCoFI's design correctly enforces CFI, adding confidence in the correctness of our system.

References

[1] AlephOne, "Smashing the stack for fun and profit." [Online]. Available: http://www.fc.net/phrack/files/p49/p49-14

[2] Solar Designer, "return-to-libc attack," August 1997, http://www.securityfocus.com/archive/1/7480.

[3] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, pp. 4:1–4:40, November 2009.

[5] J. Criswell, N. Geoffray, and V. Adve, "Memory safety for low-level software/hardware interactions," in *Proceedings of the Eighteenth Usenix Security Symposium*, August 2009.

[6] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2010.

[7] D. P. Bovet and M. Cesati, *Understanding the LINUX Kernel*, 2nd ed. Sebastopol, CA: O'Reilly, 2003.

[8] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4 BSD Operating System*. Redwood City, CA: Addison-Wesley Publishing Company, Inc., 1996.

[9] B. Zeng, G. Tan, and U. Erlingsson, "Strato: a retargetable framework for low-level inlined-reference monitors," in *Proceedings of the 22nd USENIX conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 369–382.

[10] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the 22nd USENIX conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 337–352.

[11] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: lightweight kernel protection against returntouser attacks," in *Proceedings of the 21st USENIX conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2012.

[12] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems," in *Proc. ACM SIGOPS Symp. on Op. Sys. Principles*, Stevenson, WA, USA, October 2007.

[13] C. Lattner, A. D. Lenharth, and V. S. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, USA, June 2007, pp. 278–289.

[14] J. Salwan and A. Wirth. [Online]. Available: http://shell-storm.org/project/ROPgadget

[15] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *14th USENIX Security Symposium*, August 2004, pp. 177–192.

[16] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. IEEE, 1997, pp. 65–71.

[17] I. Unified EFI, "Unified extensible firmware interface specification: Version 2.2d," November 2010.

[18] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 29–40.

[19] J. Criswell, B. Monroe, and V. Adve, "A virtual instruction set interface for operating system kernels," in *Workshop on the Interaction between Operating Systems and Computer Architecture*, Boston, MA, USA, June 2006, pp. 26–33.

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proc. Conf. on Code Generation and Optimization*, San Jose, CA, USA, Mar 2004, pp. 75–88.

[21] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '93. New York, NY, USA: ACM, 1993.

[22] The Coq Development Team, "The Coq proof assistant reference manual (version 8.3)," 2010, http://coq.inria.fr/refman/index.html.

[23] D. A. Wheeler, "SLOCCount," 2014. [Online]. Available: http://www.dwheeler.com/sloccount/

[24] *Intel 64 and IA-32 architectures software developer's manual*. Intel, 2012, vol. 3.

[25] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual Ghost: Protecting applications from hostile operating systems," in *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2014.

[26] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. New York, NY, USA: ACM, 2010.

[27] Postmark, "Email delivery for web apps," July 2013. [Online]. Available: https://postmarkapp.com/

[28] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ser. ATEC '96. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268299.1268322

[29] J. Poskanze, "thttpd - tiny/turbo/throttling http server," 2000, http://www.acme.com/software/thttpd. [Online]. Available: http://www.acme.com/software/thttpd

[30] "Apachebench: A complete benchmarking and regression testing suite. http://freshmeat.net/projects/ apachebench/," July 2003.

[31] T. O. Project, "Openssh," 2006, http://www.openssh.com. [Online]. Available: http://www.openssh.com

[32] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*, 2013, pp. 559–573.

[33] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "Rocksalt: better, faster, stronger sfi for the x86," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 395–404.

[34] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: a sandbox for portable, untrusted x86 native code," *Commun. ACM*, vol. 53, no. 1, pp. 91–99, Jan. 2010.

[35] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, "Safedrive: Safe and recoverable extensions using language-based techniques," in *USENIX Symposium on Operating System Deisgn and Implementation*, Seattle, WA, USA, November 2006, pp. 45–60.

[36] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012.

[37] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007.

[38] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010.

[39] G. Klein *et al.*, "seL4: formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009.

[40] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," in *Proc. ACM SIGOPS Symp. on Op. Sys. Principles*, Copper Mountain, CO, USA, 1995.

[41] T. Saulpaugh and C. Mirho, *Inside the JavaOS Operating System*. Reading, MA, USA: Addison-Wesley, 1999.

[42] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken, "Implementing multiple protection domains in Java," in *USENIX Annual Technical Conference*, Jun. 1998.

[43] M. Golm, M. Felser, C. Wawersich, and J. Kleinoder, "The JX Operating System," in *Proc. USENIX Annual Technical Conference*, Monterey, CA, USA, June 2002, pp. 45–58.

[44] J. Yang and C. Hawblitzel, "Safe to the last instruction: automated verification of a type-safe operating system," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010.