HCFI: Hardware-enforced Control-Flow Integrity

Nick Christoulakis
FORTH
christoulak@ics.forth.gr

George Christou
FORTH
gchri@ics.forth.gr

Elias Athanasopoulos
VU University, Amsterdam
i.a.athanasopoulos@vu.nl

Sotiris Ioannidis
FORTH
sotiris@ics.forth.gr

ABSTRACT

Control-flow hijacking is the principal method for code-reuse techniques like Return-oriented Programming (ROP) and Jump-oriented Programming (JOP). For defending against such attacks, the community has proposed Control-flow Integrity (CFI), a technique capable of preventing exploitation by verifying that every (indirect) control-flow transfer points to a legitimate address. Enabling CFI in real systems is not straightforward, since in many cases the actual Control-flow Graph (CFG) of a program can be only approximated. Even in the case that there is perfect knowledge of the CFG, ensuring that all return instructions will return to their actual call sites, without employing a shadow stack, is questionable. On the other hand, the community has expressed concerns related to significant overheads stemming from enabling a shadow stack.

In this paper, we acknowledge the importance of a shadow stack for supporting and strengthening any CFI policy. In addition, we project that implementing a full-featured CFI-enabled Instruction Set Architecture (ISA) in actual hardware with an in-chip secure memory can be efficiently carried out and the prototype experiences negligible overheads. For supporting our case, we implement HCFI by modifying a SPARC SoC and evaluate the prototype on an FPGA board by running all SPECInt benchmarks instrumented with a fine-grained CFI policy. The evaluation shows that HCFI can effectively protect applications from code-reuse attacks, while adding less than 1% runtime overhead.

1. INTRODUCTION

Exploitation of modern software is undoubtedly still possible, despite many mitigation techniques that have been enabled in production systems. Although a simple stack smashing [26] is unlikely to be sufficient for compromising a program due to non-executable data protection (DEP) [4], advanced exploitation techniques, based on code reuse, commonly known as Return-Oriented Programming (ROP) [31] and Jump-Oriented Programming (JOP) [7], are so powerful that can potentially take advantage of any vulnerability and transform it to a functional exploit. Code randomization techniques [23, 28, 30, 37] attempt to make code reuse harder by shuffling the location of the code to be reused, but it has been demonstrated that even a simple information leak can reveal all of the process' layout and essentially bypass any randomization scheme [34].

Therefore, for fighting software exploitation, the community seeks protection schemes that are based on core principles. One promising direction is based on the observation that modern exploits introduce control flows that are not part of the program's Control-flow Graph (CFG). Control-flow Integrity (CFI) [3] suggests that a running program should exhibit only the control flows that are part of the program's original CFG as expressed by its source code. Essentially, CFI mandates that any indirect branch should not be possible to target the address of *any* instruction in the program, but rather be constrained in an allowable set of addresses that have been a priori determined. For example, consider that in principle a return instruction should be *only* able to transfer control to the call site responsible for the associated function call.

CFI, although a strong principle, has still two open issues related to the technique's *accuracy* and *performance*. As far as accuracy is concerned, it is not always trivial to compute the program's CFG. This is mainly because the source code might not be always available, dynamic code might be introduced at run-time [6], and heavy use of function pointers can lead to inconclusive target resolution. This problem has led researchers to develop CFI techniques that are based on a relaxed approximation of the CFG [41, 42], also known as coarse-grained CFI. Unfortunately, coarse-grained CFI has been demonstrated to exhibit weak security guarantees and it is today well established that it can be bypassed [20].

Since approximation of the ideal CFG through code analysis does not have sound protection, at least for protecting backward edges, the community has suggested the use of a *shadow stack* [14]. A shadow stack is secure memory where, during a function call, the call site is saved. Once the function is to return, the information stored in the shadow stack is checked with the return address stored in the actual stack; in case there is a mismatch, a violation is recorded and the running process is halted. There is much criticism about the use of a shadow stack due to performance implications. However, it was recently demonstrated that even an *ideal* CFI implementation, without the use of a shadow stack, *is* vulnerable [10]. This is mainly because any CFG contains functions (e.g., `memcmp`) which are called by many different

CODASPY'16 March 9-11, 2016, New Orleans, LA, USA

 ISBN 978-1-4503-3935-3/16/03...$15.00

DOI: http://dx.doi.org/10.1145/2857705.2857735

locations of the program and essentially allow the attacker to be flexible in creating call-preceded chains of gadgets for finally exploiting the program. It is thus vital for *any* CFI implementation to employ a shadow stack.

In this paper, we acknowledge that the use of a shadow stack is mandatory for any practical CFI deployment. We further attempt to quantify the performance of CFI and demonstrate that the technique can be applied to real systems with practically negligible overhead. For proving our case, we present HCFI, a full-featured hardware implementation of CFI. We extend an existing Instruction Set Architecture (ISA), with instructions dedicated for CFI and we deploy shadow memory inside the core. We modify a SPARC SoC and evaluate the prototype on an FPGA board by running all SPECInt benchmarks instrumented with the additional CFI-related instructions. The evaluation shows that HCFI can effectively protect applications from code-reuse attacks, while adding less than 1% runtime overhead.

Compared to similar hardware implementations, such as HAFIX [15], HCFI is (i) *complete*, since it protects both forward and backward edges, (ii) *faster*, since the experienced overhead is on average less than 1%, and (iii) *more accurate*, since it employs a full-functional shadow stack implemented inside the core. Especially, as far as shadow memory is concerned, HCFI uses a novel system for supporting multiple recursive calls. Each time a return address is to be saved in the secure memory it is checked with the top of the shadow stack and if the address is matched, indicating there is a recursive call, no additional memory is wasted. This dramatically simplifies the design and reduces the space requirements, but implies that a recursive call can return to its call site immediately from any depth, thus violating a perfect CFI policy. However, we anticipate that this policy relaxation has not severe security implications, since system calls and sensitive functions are not recursive and they do not call recursive functions (i.e., hijacking a recursive function called by a sensitive system call for jumping to the sensitive call site is not possible). Furthermore, in terms of completeness, we argue that HCFI is the most rich hardware implementation of CFI so far, supporting many problematic cases (such as `setjmp/longjmp`), which we discuss thoroughly in Section 3.

1.1 Contributions

This paper contributes the following.

1. We design, implement, and evaluate HCFI, a full-featured ISA for supporting processes hardened with CFI. The prototype is based on extending a SPARC SoC and it includes a hardware implementation of a shadow stack.
2. HCFI is complete and accurate. It protects both forward and backward edges, and the shadow stack implementation can handle recursion of arbitrary depth.
3. HCFI has practically negligible overhead. We evaluate HCFI with all SPECInt benchmarks and we record a runtime overhead of less than 1% on average, which, to the best of our knowledge, stands for the first hardware implementation for full CFI support with low cost.
4. HCFI is policy agnostic and can deal with all idioms that usually interfere with hardening indirect jumps, such as the use of `longjmp` and `setjmp`. For the purpose of presenting HCFI in this paper we enable a fine-grained CFI policy with shadow-stack support.

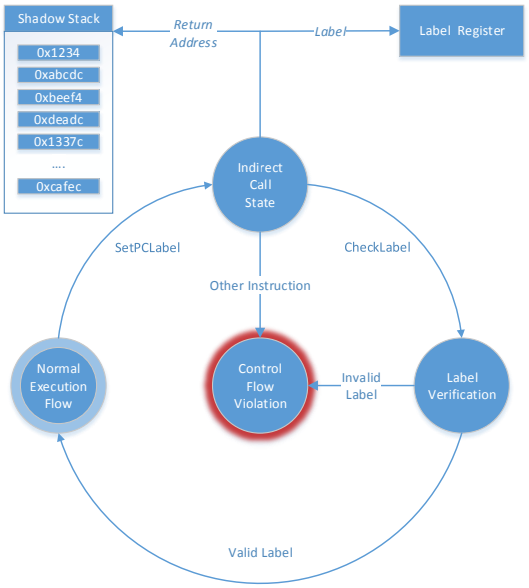

Figure 1: Indirect Call States. A SetPCLabel instruction is received, the appropriate memory modules are set, and the core enters a state where only CheckLabel instructions are accepted. Once a CheckLabel instruction is received, the labels are compared and execution returns to its normal flow.

1.2 Organization

This paper is organized as follows. In Section 2 we discuss the generic architecture of HCFI and in Section 3 we thoroughly present the technical details for implementing the prototype. We evaluate HCFI in terms of security in Section 4 and in terms of performance in Section 5. We discuss various aspects of our current and future work in Section 6. We review related work in Section 7 and, finally, we conclude in Section 8.

2. HCFI ARCHITECTURE

2.1 Control-Flow Integrity enforcement

Control-flow Integrity (CFI) aims at guaranteeing that the execution flow adheres to the path determined by the control-flow graph of the program. The control flow of a program can be manipulated either on the forward-edge, when the target of an indirect jump is altered, or on the backward-edge, when a saved return address has been changed. For forward-edges we ensure that an indirect jump can target only a function entry with the appropriate label that is generated during the CFG extraction. For backward-edges we validate that the function's return instruction targets the address of the original call site(Call-Ret pair). A more detailed discussion follows.

2.1.1 Forward-edge

The forward-edge is handled as discussed in the original CFI proposal [3]. Every indirectly called function is hard-coded with a label on its entry point. Before the indirect function call, the function's label is compared to a label assigned to the call site. Our approach differs in that we set the label before the indirect call executes, while the valida-

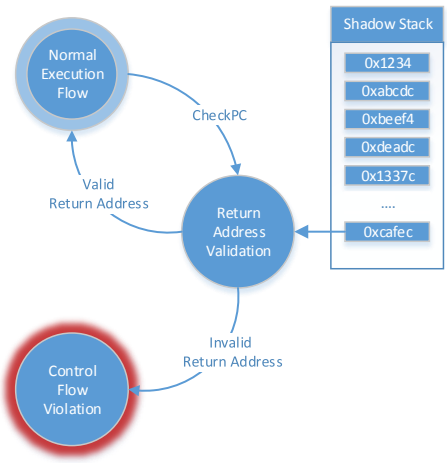

Figure 2: Return States. A CheckPC instruction is received, the Program Counter is compared with the top value of the stack and the execution continues normally.

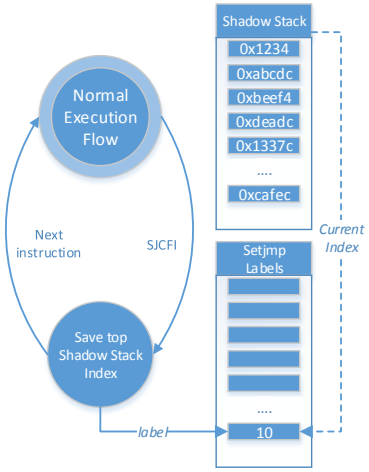

Figure 3: Setjmp Finite State Machine. An SJCFI instruction stores the current state of the Shadow Stack.

tion takes place immediately after function entry. Before the indirect control transfer, a SetPCLabel instruction, placed on the delay slot(described in subsection 3.2), stores a label in a non memory-mapped latch that resides inside the core. On the function entry, a CheckLabel instruction verifies that the label equals the one stored in the latch. If the comparison fails, a control-flow violation is detected, an exception is raised, and the system handles it appropriately.

2.1.2 *Backward-edge*

For the backward edges, a non memory-mapped stack, which also resides within the core, is deployed. The concept of a shadow stack is thoroughly studied in the literature [3, 10, 14]. The general concept is based on the notion that a function's return address points to the instruction lying directly below the call site. This is not always the case as it is common that a function does not return to the original call site.

The shadow stack of HCFI is implemented as follows. Before a call instruction executes, a copy of the return address is pushed to the shadow stack. When the callee function returns, the return address is compared with the one on the top of the shadow stack. If they are not equal, a control-flow violation is detected and handled appropriately by the system.

Notice that every direct call instruction is paired with a SetPC instruction placed on its delay slot. The SetPC instruction pushes the current Program Counter to the shadow stack module. After the callee function returns, a CheckPC instruction, placed in the delay slot of the return instruction, checks that the computed return address is equal with the address stored in the shadow stack incremented by four (one instruction below the SetPC). If the check fails, a hardware exception is raised, which is handled by the supervising firmware. An alternative way to process a mismatch between the shadow stack and the main stack, is to *silently* force the address obtained from the shadow stack as the return address. Aforesaid proposition can potentially enhance our architecture with fault tolerance capabilities, since any tampering of the return address would be rectified by the hardware.

2.2 Architecture Overview

HCFI is based on a series of modifications of Leon3 [18] core's pipeline. The architecture consists of unmapped *shadow* memory elements, more specifically a shadow stack, a shadow memory array, a shadow register, and six dedicated instructions which function upon the shadow memory elements. The shadow stack is utilized in enforcing backward-edge CFI through the detection of control-flow changes caused by arbitrary return address modifications, e.g. buffer overflows. Likewise, a single shadow register is used for enforcing forward-edge CFI, effectively protecting the execution flow from vulnerable function pointers. The shadow memory array is used for assisting setjmp/longjmp support. To access and utilize the shadow memory blocks, we extended the SparcV8 [1] instruction set with six instructions.

2.3 ISA Extension

We extended the SparcV8 ISA with six instructions designed to provide CFI functionality to the core.

SetPC: Paired with direct call instructions. The SetPC instruction is placed in the delay slot of the call instruction it is paired with. It pushes the currently executing Program Counter (PC) to the shadow stack. Also, if the next instruction is a CheckLabel, the SetPC instruction suppresses the CFI violation that would occur, since the Label Register's value has not been initialized, yet. This functionality is useful in cases where an indirectly called function is also called directly.

SetPCLabel: Paired with indirect call instructions. This instruction is placed in the delay slot of the indirect call it is paired with. Its 18 Least Significant(LS) bits carry the label used to validate the indirect call target. As with the SetPC instruction, the current Program Counter is pushed to the shadow stack. At the same time, the 18 LS bits are stored in the Label Register to be used later for validation. If the next instruction executed is not a CheckLabel, a CFI violation occurs.

CheckLabel: Placed on the entry point of a function that

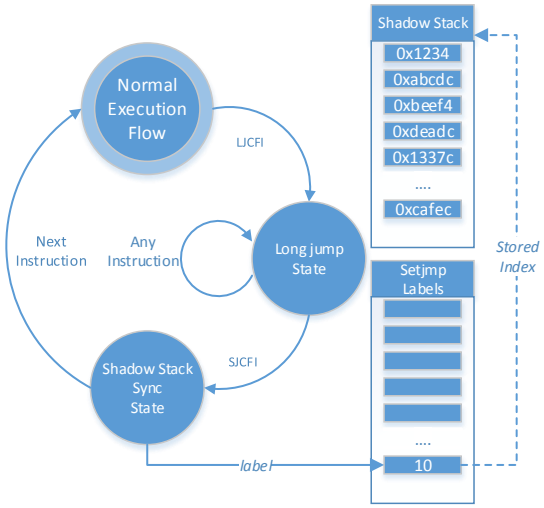

Figure 4: Longjmp Finite State Machine. An LJCFI instruction puts the core in a state waiting for an SJCFI instruction. The next SJCFI will not store the environment but restore it to the state that it was the last time an SJCFi instruction was executed on its own.

is found during instrumentation to be called indirectly. It is the only instruction that can be legally executed after a SetPCLabel. Its 18 LS bits carry the label used to validate the indirect call target. It compares the label carried on its 18 LS bits with the value stored in the Label Register. If the labels match, the register is reset and the execution continues normally, otherwise a CFI violation is detected. Since the Label Register is zeroed out after every CheckLabel, and no function is assigned zero as a label, the Label Register cannot be reused without being set again.

CheckPC: Paired with return instructions. This instruction is placed in the delay slot of the return instruction it is paired with. It compares the program counter of the next instruction executed (after the branch takes place) with the top of the shadow stack. If PC equality is confirmed, the shadow stack is popped and the execution continues. Otherwise, a CFI violation occurs.

2.4 Shadow Stack Incompatibilities

Backward-edge control-flow integrity relies on the Call-Ret pair model of programs. That means that each time a return instruction jumps to an address different than the one it was called from, the Call-Ret pair model is violated which leads to a CFI violation. Unfortunately, there are cases that Call-Ret pairs are violated in a legitimate way.

2.4.1 Setjmp/Longjmp

The most common violation is setjmp/longjmp. When a longjmp occurs, the original stack may *unwind* by several frames. Standard solutions suggest unwinding the shadow stack till it is empty or a match is found. Such an approach would impose a greater performance overhead in our design. In the proposal by Davi et al. [15], a longjmp would not cause a control-flow violation but the intermediate labels would remain active, significantly relaxing CFI. We overcome these problems by using dedicated instructions for setjmp/longjmp support, without sacrificing any security or performance.

Additionally, some designs [27] propose popping the shadow stack till a valid return address is found or it is empty, causing additional delay cycles. Furthermore, unwinding the shadow stack can lead to inconsistencies. The address required could exist more than once in the shadow stack, and since the hardware could not blindly know which of the addresses is the correct call site, it could settle on the wrong one, causing a violation later in the execution. Other proposals do not support setjmp/longjmp functionality and any such jump would be perceived as a control-flow violation.

In order to achieve *on cycle* synchronization between the shadow stack and the normal stack, we decided on the addition of two dedicated instructions and a shadow memory array. Those new instructions are paired with the call instructions to the setjmp and longjmp functions themselves.

SJCFI: The first one, SJCFI, is paired with the setjmp function. It is placed two instructions below the call to setjmp - the instruction to which a call to setjmp would return to. It carries a unique label on it 8LS bits - different from the label used for forward edge enforcement. Much like setjmp, it serves two purposes, (i) it sets the environment to support a longjmp, and (ii) acts as a landing point for the jump itself.

In the first case, once a setjmp returns, the first instruction executed would be SJCFI. The label is used as an index to the new memory element. During SJCFI's execution, the index of the top element of the shadow stack is stored in the new memory component using the label as an index. This allows pairing this particular landing point to the current state of the shadow stack. Even though the addresses remain in the shadow stack, they cannot be exploited by an attacker as the only way to use them would be to raise the index, which cannot happen without overwriting the addresses with correct ones.

SJCFI also acts as a landing point for a longjmp. Since it is placed two instructions below the call to setjmp, and a longjump will return to its equivalent setjmp call site, it will be the first instruction executed after such a jump.

For SJCFI to support long jumps, an LJCFI instruction is assumed to have been already executed. In this case, SJCFI, instead of reading the index of the stack and writing it to the new memory element, reads the index from the memory element (once again using its label) and sets the stack to it. Since the index of the shadow stack corresponds with the stack frame once again, execution and CFI enforcement can continue normally. The next SJCFI instruction executed will use the first functionality unless another LJCFI was executed before that.

LJCFI: LJCFI instruction is only used to signify that a longjmp is underway. It is placed in the delay slot of a longjmp call and flags that a longjmp is executed. After the longjmp function is executed, the program counter should point to an SJCFI instruction, which will use the second of its functionalities, synchronizing the shadow stack and clearing the longjmp state flag.

The functionality of those two instructions is graphically represented in figures 3 and 4.

2.4.2 Tail-Call Elimination

Another case of Call-Ret pair violation is *tail call elimination*. As shown in figure 7, before calling *bar*, *foo's* return address (stored inside the o7 register) is *moved* to global reg-

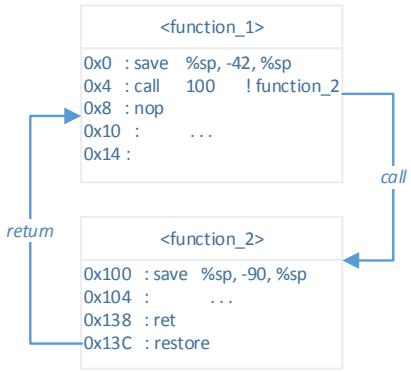

Figure 5: SparcV8 assembly - direct function call without CFI instrumentation.

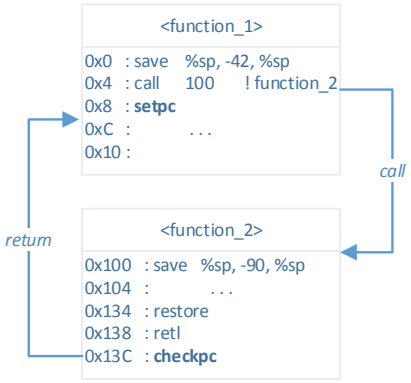

Figure 6: SparcV8 assembly - direct function call with CFI instrumentation. A SetPC instruction is placed on the delay slot of the call instruction, and a CheckPC on the delay slot below the return. The restore instruction is pushed above the return and the return instruction changes to account for it.

ister g1. When the call instruction is executed, register o7 will be overwritten with the current program counter (0x20) which serves as *bar* function's return address. Finally, in the delay slot of the call instruction, the return address of *foo* is restored in register o7. The effect of the above code snippet is that *bar* function will return to *foo* function's call site. In our design, this optimization renders the shadow-stack inconsistent with the main stack. Thus, this particular optimization has to be disabled in order to run the benchmarks. Adding support for this optimization is possible, by simply not instrumenting the eliminated call site, though that might pose a great security concern.

2.5 Recursion Support

The memory available to the shadow stack is finite and implemented statically inside the core, where there is no dynamic memory allocation. Therefore, for supporting recursion, additional features should be implemented.

Before the SetPC and SetPCLabel instructions push the current PC to the shadow stack, the stack is topped and the two addresses are compared. If the addresses are different, the new address is pushed. Otherwise, the address is not pushed, but the current index of the shadow stack is marked as *recursive* on a separate 128*1 bitmap.

Figure 7: SPARC V8 tail call elimination example. The calling function (foo) replaces the return address of the callee function (bar) with its own. Bar will return to the function that called foo, skipping it.

During the CheckPC execution, if the address currently being compared has the recursion bit activated, it is not popped from the stack. If the address comparison results in a mismatch and the top address is recursive, the top address is popped and the PC is compared with the next one. If the addresses match, execution continues normally and, if the (now) top address is not marked as recursive, it is popped. Otherwise, should the comparison again result in a mismatch, the corresponding violation is raised.

2.6 Instrumentation

Instrumentation takes place at the assembly-level of C programs using a Python script (about 200 LoCs). We assume that input programs are products of standard C compilers (such as, GCC and Clang), and they do not include custom assembly idioms. Instrumentation by no means is limited to C-compiler generated assembly. In fact, any assembly code is instrumentable as long as a Call-Ret-like model is sustained.

In figures 5 and 6 we show the instrumentation of a direct call in the SparcV8 assembly language. The logic behind the instrumentation is fairly simple as it consists of pairing every call and return with a CFI instruction. For calls, a SetPC instruction is added below them, and for returns, a CheckPC.

The new instructions were designed to take advantage of the delay slot below branches in the SparcV8 architecture. With that in mind, any instructions residing in the delay slot must be moved out of the slot, before the branch. The most usual case of instructions that need to be moved are *restore* instructions as they almost always reside in the delay slot of their respective return instructions.

Since the *restore* instruction changes the focus of the register window, we must compensate for it moving before the return instruction. The *ret* instruction expects to find the return address in register i7, but because the register windows have shifted, the appropriate value is now stored in register o7. Thankfully, the Sparc assembler provides another instruction with this case in mind, *retl*.

In Figures 8 and 9 we show the instrumentation of an indirect function call in the SparcV8 assembly language. The backward-edge components remain essentially the same, with the only modification being that the SetPC instruction is switched with a SetPCLabel instruction. But now, the forward-edge components are also in use. Specifically, SetPCLabel, storing the hard-coded label for later comparison,

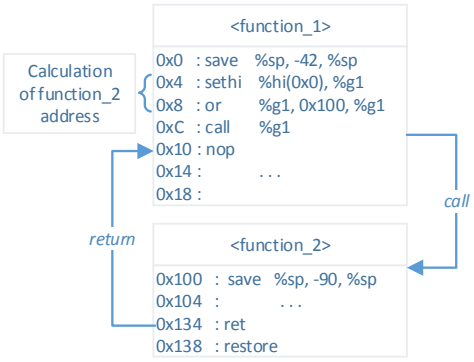

Figure 8: SparcV8 indirect function call without CFI instrumentation. The address in loaded in a register which is used to perform the indirect call. Otherwise, the call performs similarly to the direct call.

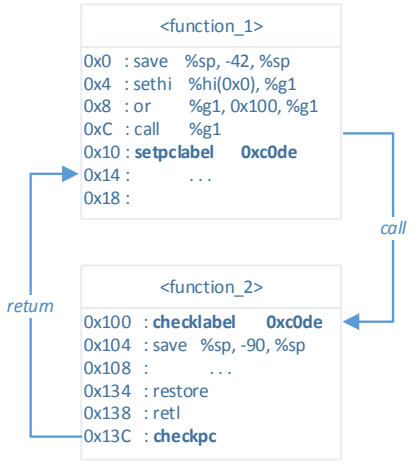

Figure 9: SparcV8 indirect function call with CFI instrumentation. A SetPCLabel instruction is placed on the delay slot below the indirect call. A CheckLabel instruction is placed on the entry point of the indirectly called function. Finally, a CheckPC instruction is placed in the delay slot of the return instruction.

and the CheckLabel instruction, placed on the function entry point, performing said comparison.

3. HCFI PROTOTYPE IMPLEMENTATION

In this section we describe the HCFI prototype implementation, we present the results of the hardware synthesis using an Virtex 6 [40] FPGA board, in terms of additional hardware needed compared to the unmodified processor, and finally we discuss how the proposed system can be easily ported to other architectures and systems.

3.1 Introduction to the Leon3 Softcore

We modified the Leon3 SPARC V8 processor [18], a 32-bit open-source synthesizable processor, to implement the security features required for a hardware-based CFI support. All hardware modifications require less than 500 lines of VHDL code. Leon3 uses a single-issue, 7-stage pipeline. Our implementation has 8 register windows, an 16 KB 2-way set associative instruction cache, and a 16 KB 4-way set associative data cache.

3.2 Delay Slot

In the SparcV8 architecture, as with many other RISC ISAs, exists the concept of a delay slot. In those architectures, any instruction directly below a branch is always executed as if before it, regardless of the result. Subsequently, the instruction slot below a branch is called a delay slot. HCFI was built with that mechanism in mind, though it is by no means a prerequisite.

3.3 Memory Element Additions

The implementation of the prototype presented in this paper requires several memory elements. Specifically, a dedicated 32 bit register, a dedicated 128*32 bit stack, a bitmap of 128 bits, and a dedicated 128*8 bit memory module. All memory elements are only accessible using the new CFI instructions of the prototype.

The register (Label Register) is used in storing the label used for indirect jump verification - forward edge. The stack (Shadow Stack) is used in storing the return addresses of the functions currently executing, so as to add a measure of redundancy and validate return instructions - backward edge. The bitmap holds the recursion bit for the return addresses of the Shadow Stack. The third memory module is used to provide setjmp/longjmp support.

All four memory elements are are not *memory-mapped*, and thus are only accessible through the use of the CFI instructions, while there is no interference with additional peripherals or supervising software. Since the memory elements do not rely on the data cache, or use any existing buses, they do not encumber the core's memory bandwidth. Also, since the elements do not reside in RAM, they can be accesed with just one cycle of delay for both reads and writes.

3.4 Leon3 Pipeline Modifications

The modifications required for supporting the new instructions, discussed in Section 2, to the core are exclusive to the pipeline. The design relies on a new `process`, the hardware equivalent of a software thread, for avoiding heavy modifications to the critical path of the pipeline. The process contains all the CFI functionality, while the Leon3 pipeline is only modified to handle the input and output for the process; such as the current and next Program Counter, signals indicating annulled instructions, exceptions, and the instructions themselves. We discuss here how each instruction is implemented.

SetPC: The basic function of the SetPC instruction is to push the current PC to the Shadow Stack during the memory stage of the execution. Additionally, during the execution stage of the pipeline, it sets a flag that is used to suppress the `Invalid Label` violation (discussed in subsection 3.5) that occurs if the next instruction executed is a CheckLabel. If the next instruction is not in fact a CheckLabel, the flag is reset. This implementation allows a function called directly to be called indirectly, as well. To avoid an exception, the violation must be suppressed, as the Label Register is not currently set.

For supporting recursion, the instruction first tops the stack during the register access stage of the execution. If the address is the same as the current PC, it does not push it to

the stack but instead marks the current index as recursive. Otherwise, it performs as previously described.

SetPCLabel: This instruction also pushes the current PC to the Shadow Stack during the memory stage and supports recursive calls as SetPC does. Additionally, SetPCLabel sets the Label Register to the value carried in its 18 LS bits. The value is extracted from the instruction during the decode stage and is set to the Label Register during the memory stage of the execution. Finally, it forces a check that ensures that the next instruction executed is in fact a CheckLabel. If the next instruction is not a CheckLabel, then a violation is raised that will lead to an exception during the violating instruction's exception stage.

CheckPC: The CheckPC instruction serves a simple purpose. During the register access stage, it *tops* the Shadow Stack, increments the value by 4 (one instruction below the SetPC), and compares the result with the next Program Counter (nPC). If equality is confirmed, then the stack is *popped*. If the result is not the expected value, a violation is raised leading to an exception during the exception stage.

Much like the SetPC and SetPCLabel instructions, if recursion optimization is in place, the functionality shifts. If the top address in the stack is marked as recursive, it is not popped, so that it can be used again later. If the address comparison results in a mismatch and the top address is marked as recursive, the stack is popped and another comparison is performed two cycles later, during the memory access stage. If the new comparison holds, execution continues normally and, if the top address is not recursive, it is popped. If the comparison fails again, a mismatch violation is raised during the exception stage.

CheckLabel: This instruction, much like the SetPCLabel instruction, carries a label on its 18 LS bits. This label is extracted during the decode stage of the execution, and compared to the label stored in the Label Register during the execution stage. If label equality is not confirmed, then a violation is raised leading to an exception. The Label Register is reset during the memory stage.

The CheckLabel instruction requires that a SetPC or SetPCLabel instruction was the last instruction to execute. Otherwise the Label Register is not set and its contents are zeroed. This leads to a violation, as no function is assigned zero as a label, unless a SetPC is the last instruction executed, which suppresses the violation.

LJCFI: LJCFI raises a flag to signify that a longjmp is underway. It does not carry any labels or uses any memory beyond the signal used for the flag.

SJCFI: SJCFI carries a label in its 8LS bits that is extracted at the decode stage. During the execution stage, depending on whether the flag is set by LJCFI, it either reads the top value's index from the Shadow Stack or retrieves the new index from the new memory element, using the label as a pointer to it. Finally, during the memory stage, again depending on the flag, it either stores the Shadow Stack's index to the memory element with the label as a pointer, or it sets the index retrieved from the new memory element to the Shadow Stack.

3.5 Violations

The various problems and errors detected during execution are summed in the following violations:

Label Mismatch: Raised when the label stored in the Label Register is not equal to the label carried by the CheckLabel instruction. It can also mean that the Label Register has not been set at all. This is a forward-edge CFI violation.

PC Mismatch: Raised when a CheckPC instruction detects tampering on the return address. The address stored in the Shadow Stack is not the address to which the return instruction jumped. This is a backward-edge CFI violation.

Flow: Raised when the instruction executed after a SetPCLabel is not a CheckLabel. The indirect call targeted a function that has not been found to be a valid indirect target during instrumentation. This is a forward-edge CFI violation.

Empty: Raised when a CheckPC instruction tries to validate a return address while the stack is empty. More return addresses have been popped than have been pushed. This is a backward-edge CFI violation.

Full: Raised when a SetPC or a SetPCLabel instruction pushes a return address while the stack is full. This is not a CFI violation, but an error that is raised when the stack fills. For the implementation presented in this paper, a 128 word Shadow Stack is used and is capable to run all benchmarks. Nevertheless, a larger Shadow Stack can be easily placed in the core if needed.

In the prototype implemented on the Leon3 softcore, all violations are designed to lead to an illegal instruction exception that puts the Integer Unit in Error Mode thus halting the execution. Alternatively, a custom exception can be easily created and handled by either the hardware or the supervising software.

3.6 Portability to Other Architectures

The design of our implementation does not actively change the core's architecture, but simply adds a few components and checks. The design only touches on very basic concepts of computer architecture, like the Program Counter, interrupts and exceptions, that are present in any modern core. All modifications for supporting HCFI are only additive to the processor, and rely on components present in any architecture. Therefore, the design presented in this paper can be ported to any architecture with minimal effort, and, as shown in section 5.5, with a small area overhead footprint.

4. SECURITY EVALUATION

In this section we discuss the security guarantees provided by HCFI. In our threat model we assume that the attacker can exploit a vulnerability, either a stack or heap overflow, or use-after-free, present in the application's source code. This vulnerability can be further used to overwrite key components of the running process like return addresses, function pointers, or VTable pointers. We also consider that the attacker has successfully bypassed ASLR or fine-grained randomization [34], and has full knowledge of the process' memory layout. Nevertheless, the system enforces that (i) the `.text` segment is non-writable preventing the application's code from being overwritten, and (ii) the `.data` segment is non-executable [4] blocking the attacker from executing directly data with proper CFI annotation. Both of those principles are commonplace in today's systems preventing software exploitation.

4.1 Defence with CFI ISA extensions

By forcing every return instruction to adhere to the address stored at the top of the Shadow Stack, ROP attacks are effectively foiled. In all our tests, every change in the control flow of the application, provoked by a return instruction that was not consistent with the Shadow Stack's top value, led to a CFI violation being raised, leading to a trap in the core and the eventual termination of the execution.

Similarly, an indirect call not leading to a pre-approved function entry point would always raise a CFI violation and halt the execution. Thus, foiling again most JOP attacks by limiting the possible positions in the program that such a jump would be allowed to target. The granularity of the forward-edge protection is directly proportional to the depth of the analysis performed on compile time.

4.2 Efficacy

We run a multitude of small programs designed to violate the CFI principles in different ways, e.g. indirectly jumping with invalid labels, or no labels at all, modifying return addresses on runtime, stressing the Shadow Stack, and various others. Using behavioural simulation with Xilinx's [39] Isim tool, we had total transparency of every signal in the Leon3 softcore, and therefore the shadow memory elements themselves. We could observe every microbenchmark's effect on the Shadow Stack and the core in general. The observations were consistent with our expectations. Every control-flow violation expected was raised and detected, halting the execution. Finally, we further confirmed our observations by additonally running the microbenchmarks on the programmed FPGA board, again finding the expected results.

5. PERFORMANCE EVALUATION

5.1 Testing Environment

We synthesized and programmed the modified Leon3 softcore on a Xilinx ml605-rev.e FPGA board. The FPGA has 1024 MB DDR3 SO-DIMM memory and the design operates at 120 MHz clock frequency. It has also several peripherals including an 100Mb Ethernet interface. Since we are targeting embedded systems, we ran all tests without an operating system present. The benchmarks are SpecInt2000 [35] and a few microprocessor-based, namely Coremark [17], Dhrystone [38], and matmul [9].

5.2 BareFS

Since the Spec suite is not designed for use in embedded systems, and running on bare-metal has the drawback of not offering the functionality of either files or command line arguments, we had to modify the code of each benchmark in order for it to be able to read its input files and arguments. The modifications included hard-coding all input files and required command line arguments to buffers, as well as changing any instructions related to I/O so that all input comes from memory, and any possible output is either discarded, written to a new buffer, or sent to stdout/stderr.

We automated the modification process by creating a library, which overloads a large part of the standard library and a python script that, given the input files, creates a C file containing buffers initialised to the contents of the files. The library redefines all standard function calls (such as open, fopen, fscanf, fgets, etc) with custom ones.

5.3 NOP Equivalence & Profiler Verification

Due to the architecture of the Leon3 core, our CFI instructions have the same execution time as a NOP instruction. This allows us to perform various sanity checks during our testing phase, with regards to expected overhead. One such test consists of running all benchmarks, on an unmodified (vanilla) Leon3 core, with NOP instructions in place of our CFI instructions. All checks performed during the testing phase verified our results. Finally all results are also verified by using a profiler to count all calls, both direct and indirect, and returns executed during the benchmarks' runtimes. Again, all results are consistent.

5.4 Runtime Overhead

To measure the overall runtime overhead we run for multiple times each benchmark, instrumented with CFI instructions, on the modified core, which is programmed on the ml605-rev.e FPGA. Before each run, both the instruction and data cache are flushed. The results are depicted in Figure 10 and the runtime overhead is under 1%.

We have omitted gcc and eon from SpecInt2000. In the case of gcc, CFI violations occur during normal execution, since several return addresses change after being pushed to the shadow stack. This has been confirmed by Dang et al. [14]. For evaluating gcc we count NOP instructions, since they are equivalent to CFI instructions (see Section 5.3). While the overhead reported is without the full CFI instrumentation, counting NOP instructions is really close to measuring the actual CFI instrumentation.

We are also unable to sufficiently instrument eon (written in C++). The main problems are that we could not detect VTables and that some return addresses changed during runtime. An analysis of the code, on the assembly level, revealed that the program loaded return addresses from memory, a few stack frames below the current one.

Interestingly, the gap benchmark came very close to reaching the maximum theoretical overhead of 6.60%; measured by running the worst case scenario - a loop executing only indirect calls to a function, which, in turn, immediately executes a return instruction.

5.5 Hardware Overhead

We implemented our design firstly without setjmp/longjmp support or the recursion optimization. The resulting area overhead of our implementation, as detailed by the reports of the Xilinx tools used to synthesize the design, was very low, using an additional 0.65% registers and 0.81% LUTs (look-up tables). With setjmp/longjmp support and the recursion optimization in place, the area overhead increased significantly to 2.52% registers and 2.55% LUTs. The additions to the design do not seem to add to the critical path of the processor and thus do not lower the maximum frequency that the core can achieve on the board.

6. DISCUSSION & FUTURE WORK

HCFI's design does not offer support for multi-threaded environments. A single shadow stack located in the core is not sufficient to store the return addresses for all the processes that share the processor. Implementing an array of shadow stacks inside the core would be a step towards achieving this functionality. Unfortunately, such a hardware implementation is not feasible, due to the substantial area overhead it would introduce; the array would have to be

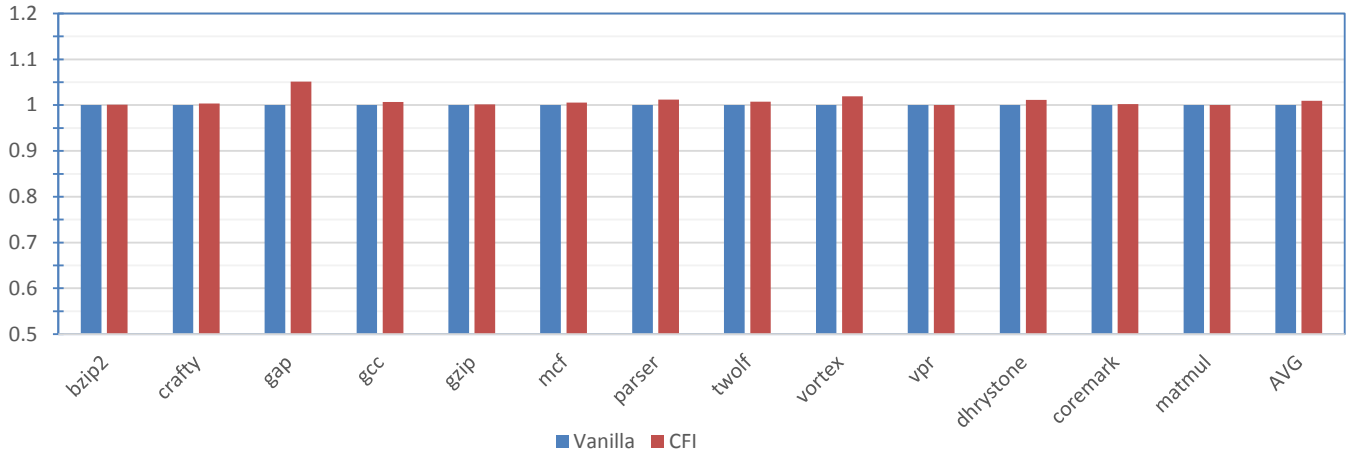

Figure 10: Presentation of the runtime overhead measured with our implementation compared to the runtime on a vanilla Leon3 core.

large enough to store the shadow stack for every active process.

This approach can be easily implemented by having a small array of shadow stacks indexed by the processes' IDs. When a context switch occurs, the operating system has to store the new running process' ID to a memory-mapped register that is visible to the control-flow integrity pipeline. The pipeline would, in turn, use it to select the appropriate shadow stack for the running process. When the process is terminated, a *cfiexit* instruction will be issued in order to invalidate the process' slot in the shadow stack array. Additionally, several software stacks could be integrated in one hardware stack. Instead of using an array of shadow stacks, the operating system could use one shadow stack for storing return addresses along with their respective process ID. Once a process terminates, all stale records contained at the shadow stack should be cleaned up by the operating system. Enabling multiple shadow stacks is part of our future work.

The most novel approach, that handles multi-threaded environments, is proposed by IAD [2]. They conclude that the optimal design for a shadow memory assisted architecture has to derive a large subset of the logic from the existing MMU subsystem. In essence, the core is augmented with a second MMU(Shadow MMU), that reserves a small part of the system's memory, and denies access to it for everything but itself and the the set of CFI instructions from the core's extended ISA.

Compared to our design, the above approach could provide the same level of security, but would trade-off performance for multi-threading support. The performance degradation is owed to the fact that the CFI instructions would now operate on the *slow* system memory, and not the dedicated memory located inside the processor's core. An obvious optimization, is to include enough memory for one process inside the core, much like in our current design, and utilize the shadow MMU in order to swap the CFI values of the executing thread at every context switch. With this optimization, the shadow MMU architecture would add minor performance overhead over our initial design, considering the overhead imposed by the context switch itself, and therefore it could be essentially used on top of HCFI. We plan to explore this integration in our future work.

7. RELATED WORK

Many mitigation techniques for protecting software are based on CFI. Most of them are software-based, although there are some attempts for delivering CFI-aware processors. In this section, we discuss a representative selection of CFI solutions and their limitations.

BinCFI [42] and CCFIR [41] are the first CFI implementations that can transparently work with binaries, where the source code is absent. Both techniques are based on relaxing the CFG a process should adhere to, by delivering coarser-grained CFI polices compared to the original CFI proposal [3]. Although coarse-grained CFI, as implemented in both BinCFI and CCFIR, is practical and aims at protecting directly binaries with really low overhead, the security implications cannot be overlooked, since it has been demonstrated that both techniques can be bypassed [20]. In a similar fashion, coarse-grained CFI techniques have been enabled using particular hardware features, such as the Last-branch Record (LBR) [13, 29], however, again, it was quickly demonstrated that these policies are vulnerable as well [11, 16, 21, 33].

Since coarse-grained CFI provides limited security, the community has further focused on (a) more fine-grained policies that can be applied at the compiler level and on (b) securing just the frequently exploited elements of a running process, and not all indirect branches, as for example are VTable pointers. For example, researchers have proposed techniques for securing VTables in binaries [5, 12, 19], however, since recovering the semantics of all C++ objects from binaries is not always possible, these systems are all subject to more sophisticated attacks [32]. In another direction, VTV [36], ShrinkWrap [22], and SafeDispatch [24] apply fine-grained CFI policies at the compiler level for protecting VTables. However, even fine-grained policies have been also demonstrated vulnerable, unless they include a shadow-stack implementation [10].

In this paper, we do not attempt to promote a new CFI flavor, but, rather, to argue that a full-featured system that supports fine-grained CFI policies *and* an in-chip shadow stack can be implemented and offer strong security guarantees at a low cost (less than 1% overhead on average). Similar processors have been proposed in the literature, however

none is as complete and as fast as ours. In Branch Regulation [25], neither forward or backward edge control-flow changes are secured adequately. Forward edges are augmented by coarse-grained CFI, which allows for branching to any function entry point or any point within the currently executing function. While backward edges are protected by a shadow stack that keeps track of the program's return addresses, the stack itself is not secured against tampering as it resides in mapped memory. HCFI implements a shadow stack that is *never* mapped on the host's memory.

Davi et al. [15] propose HAFIX, a system for backward edge CFI and, unlike Branch Regulation, HAFIX does use dedicated, hidden memory elements for storing critical information. Their implementation utilizes labels to mark functions as active call sites. However, this has the disadvantage of allowing the attacker to jump to any active function. This is important, since their method also allows for an attacker, using stack unwinding, to avoid the execution of CFIDEL instructions and eventually mark every function as an active one, thus effectively permitting jumps anywhere in the program, and therefore being possibly vulnerable.

Budiu et al. [8] propose the usage of hard-coded labels for both forward edge and backward edge control-flow integrity enforcement. The usage of labels for backward-edge protection certainly limits the attacker, but still allows him to take advantage of functions that are called by many call sites, such as `memcpy`. Essentially, this implementation is vulnerable, since it lacks of a shadow-stack implementation.

Finally, NSA's proposal on hardware CFI [2] facilitates a shadow stack to protect return addresses and landing point instructions to augment indirect-call transfers. In order to improve the flexibility of the shadow stack, they propose the use of a shadow MMU that will handle the management of the shadow memory. However, their proposal lacks granularity on forward-edge flow integrity, thus an attacker can point an indirect branch on any landing point instruction.

8. CONCLUSION

Many CFI policies have been proposed in current literature, however all of them have been demonstrated to be vulnerable. Recently, it was argued that even fine-grained CFI policies can be exploited, unless the policy is supported by a shadow stack. In this paper, we acknowledge that the use of a shadow stack is mandatory for any practical CFI deployment. We further attempted to quantify the performance of CFI and demonstrated that the technique can be applied to real systems with practically negligible overhead. For supporting our case, we presented HCFI, a full-featured hardware implementation of CFI. We extended an existing ISA by adding CFI assisting instructions and we deployed shadow memory inside the core. We modified a SPARC SoC and evaluated the prototype on an FPGA board by running all SPECInt benchmarks instrumented with the additional CFI-related instructions. The evaluation showed that HCFI can effectively protect applications from code-reuse attacks, while adding less than 1% runtime overhead. Compared to similar hardware implementations, HCFI is (i) *complete*, since it protects both forward and backward edges, (ii) *faster*, since the experienced overhead is on average less than 1%, and (iii) *more accurate*, since it employs a full-functional shadow stack implemented inside the core.

Acknowledgments This work was supported by the European Commission through the project SHARCS under Grant Agreement No. 644571.

9. REFERENCES

[1] The SPARC Architecture Manual, Version 8. www.sparc.com/standards/V8.pdf.

[2] Hardware Control Flow Integrity for an IT Ecosystem. https://github.com/iadgov/Control-Flow-Integrity/tree/master/paper, 2015.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[4] S. Andersen and V. Abella. Changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, Data Execution Prevention. Microsoft TechNet Library, September 2004. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[5] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[6] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis. The devil is in the constants: Bypassing defenses in browser jit engines. In *NDSS*. The Internet Society, 2015.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[8] M. Budiu, U. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 42–51. ACM, 2006.

[9] J. Burkardt, P. Puglielli, and P. S. Center. Matmul: An interactive matrix multiplication benchmark. *degrees from BITS, Pilani. He is a Fellow of the Institution of Engineers (India), Fellow of National Academy of Engineering (FNAE), Fellow of National Academy of Sciences (FNASc), Life Member ISTE(LMISTE). Professor Kothari has published/presented*, 640, 1995.

[10] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., Aug. 2015. USENIX Association.

[11] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, Aug. 2014. USENIX Association.

[12] Chao Zhang, Chengyu Songz, Kevin Zhijie Chen, Zhaofeng Cheny, and Dawn Song. Vtint: Protecting virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[13] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for

defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*, 2014.

[14] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security, ASIACCS*, volume 15, 2015.

[15] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. Hafix: hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference*, page 74. ACM, 2015.

[16] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, Aug. 2014. USENIX Association.

[17] EEMBC. Coremark Benchmark. https://www.eembc.org/coremark/.

[18] Gaisler Research. Leon3 synthesizable processor. http://www.gaisler.com.

[19] R. Gawlik and T. Holz. Towards automated integrity protection of c++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 396–405, New York, NY, USA, 2014. ACM.

[20] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 575–589. IEEE, 2014.

[21] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA, Aug. 2014. USENIX Association.

[22] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. Shrinkwrap: Vtable protection without loose ends. In *ACSAC*, pages 341–350. ACM, 2015.

[23] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 571–585, Washington, DC, USA, 2012. IEEE Computer Society.

[24] D. Jang, Z. Tatlock, and S. Lerner. Safedispatch: Securing c++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[25] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 94–105. IEEE, 2012.

[26] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):365, 1996.

[27] H. Özdoganoglu, T. Vijaykumar, C. E. Brodley, B. Kuperman, A. Jalote, et al. Smashguard: A hardware solution to prevent security attacks on the function return address. *Computers, IEEE Transactions on*, 55(10):1271–1285, 2006.

[28] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society.

[29] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, Washington, D.C., 2013. USENIX.

[30] PaX Team. Address Space Layout Randomization (ASLR), 2003. http://pax.grsecurity.net/docs/aslr.txt.

[31] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[32] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[33] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-rop defenses. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pages 88–108, 2014.

[34] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, May 2013.

[35] Standard Performance Evaluation Corporation (SPEC). SPEC CINT2000 Benchmarks. http://www.spec.org/cpu2000/CINT2000.

[36] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc and llvm. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 941–955, Berkeley, CA, USA, 2014. USENIX Association.

[37] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM.

[38] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[39] Xilinx. ISE Simulator (ISim). http://www.xilinx.com/tools/isim.htm.

[40] Xilinx. Xilinx Virtex 6 ml605 rev-e Evaluation Board. http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf, 2012.

[41] 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*, pages 559–573. IEEE, 2013.

[42] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Usenix Security*, pages 337–352, 2013.