Strategy Without Tactics: Policy-Agnostic Hardware-Enhanced Control-Flow Integrity

Dean Sullivan[1], Orlando Arias[1], Lucas Davi[2], Per Larsen[3], Ahmad-Reza Sadeghi[2], and Yier Jin[1]

[1]University of Central Florida, USA

[2]Technische Universität Darmstadt, Germany

[3]University of California, Irvine, USA

ABSTRACT

Control-flow integrity (CFI) is a general defense against code-reuse exploits that currently constitute a severe threat against diverse computing platforms. Existing CFI solutions (both in software and hardware) suffer from shortcomings such as (i) inefficiency, (ii) security weaknesses, or (iii) are not scalable. In this paper, we present a generic hardware-enhanced CFI scheme that tackles these problems and allows to enforce diverse CFI policies. Our approach fully supports multi-tasking, shared libraries, prevents various forms of code-reuse attacks, and allows CFI protected code and legacy code to co-exist. We evaluate our implementation on SPARC LEON3 and demonstrate its high efficiency.

1. INTRODUCTION

Defending against code-reuse attacks (CRA) is currently a highly active research area. Control-flow integrity (CFI) has been proposed as a general defense against control-flow hijacking attacks [3]. In particular, it defends against modern CRAs such as return-oriented programming (ROP) [19]. These attacks are prevalent, Turing-complete, and repeatedly leveraged to compromise applications.

CFI mitigates these attacks by ensuring that an application follows a legitimate control-flow path. The legitimate paths are manifested in the application's control-flow graph (CFG) derived during an offline static analysis phase. Whenever an attacker attempts to subvert the execution to follow an illegal control-flow path, CFI detects this malicious control flow, and immediately terminates the process.

The majority of CFI defenses focus on software-based implementations, where code is instrumented to incorporate control-flow checks before each indirect branch instruction. These solutions, however, suffer from performance issues as their CFI policy's precision increases. Some approaches have focused on improving performance, which has resulted in overhead being lowered [26] at the cost of security. Recent research [14, 7, 13] has highlighted that CFI policies which trade security for performance can be bypassed. Furthermore, the protection offered by *ideal* CFI has recently been questioned [6].

Ensuring the integrity of CFI-related data is similarly challenging and costly. Many CFI schemes maintain a shadow stack for return addresses to validate whether a function's return address has been corrupted by an attacker [3]. This stack needs dedicated maintenance and requires protection from the main program incurring additional overhead [11].

To tackle these shortcomings, recent CFI schemes follow hardware software co-design principles: HAFIX [12] deploys dedicated CFI instructions to enforce efficient CFI enforcement for function returns. However, it does not provide CFI enforcement of indirect jumps and calls. Also, to the best of our knowledge, there does not exist any hardware-based CFI scheme that applies to complex code including shared libraries, and supports multi-tasking as well as inter-operation with legacy code. Our approach supports these features without sacrificing performance or security. Hardware-based signature detection [15] or control data isolation [4] support these features, but are orthogonal approaches.

Contributions. In this paper, we introduce a hardware-enhanced CFI platform that scales to the coverage provided by any Control Flow Graph (CFG), enables highly efficient enforcement of diverse CFI policies, and losslessly enforces any provided CFG. We offer protection scalable to the precision of the control flow analysis, which in certain cases may be coarse and in others precise. Our platform handles shared libraries due to compiler supported ISA extensions and incorporates features to handle multi-tasking and interoperation with legacy programs unprotected by CFI. We evaluate runtime attacks and CFI vulnerabilities using hardware-enhanced CFI by first evaluating its effectiveness against the most current code-reuse attacks [6, 20, 14] affecting C and C++ applications. These attacks are able to perform malicious actions while adhering to the restrictions imposed by a CFI-protected system. We examine the runtime performance overhead of our platform using the SPEC2006 bencharks and CoreMark microbenchmarks on the SPARC LEON3 processor [18] demonstrating a negligible performance overhead; on average only 1.75% for SPEC and 0.5% for CoreMark. Finally, using Design Compiler's 32/28 nm process library, we show our hardware-enhanced CFI area overhead is negligible and that it can be clocked up to 3 GHz.

In summary, our core contributions are as follows.

1. **Scalable, Lossless CFI Enforcement**: We present a design that scales to any CFG provided and losslessly enforces the provided CFG. Our platform features new CFI instructions supporting CFI on diverse CFGs.
2. **Comprehensive Prevention**: Our CFI hardware platform prevents many known code-reuse attacks: traditional Return Oriented Programming (ROP) [19], ROP without returns [8], and full-function reuse [20].
3. **CFI Hardware Platform**: We present the design, implementation, and evaluation of an efficient hardware CFI platform on the open source LEON3 SoC.

DAC '16, June 05 - 09, 2016, Austin, TX, USA
Copyright 2016 ACM ACM 978-1-4503-3520-1/15/06 ...$15.00
http://dx.doi.org/10.1145/2744769.2747942.

4. **CFI Operating System Support**: We tackle practical challenges such as support for multitasking, dynamic linking, CFI compilation, protection of shared libraries, and co-existence of CFI and legacy code.

We stress that the goal of this paper is the design of a hardware CFI framework that can enforce CFI policies of different precision based on the CFG. Our paper is explicitly not about static analysis of source code or advanced binary analysis to extract CFGs. Generation of CFGs for real-world software remains an open research problem and issues in CFG generation are orthogonal to the challenges we address: making CFI enforcement efficient by adding dedicated instructions and supporting hardware.

2. THREAT MODEL & REQUIREMENTS

Threat Model. Our threat model follows the traditional CFI model. We assume an adversary who has arbitrary read and write access to data memory, and read access to code memory. The attacker can either be a local or remote attacker. However, the attacker only has access to user applications, as kernel exploits can undermine any security mechanism implemented for user-space applications.

CFI ensures the integrity of the program's control flow. Consequently, we target benign applications that an attacker attempts to compromise, but do not protect against applications that are inherently malicious. This includes cases where the attacker modifies the binary either in disk or memory. Further, we focus on CRAs, but not code injection attacks that are prevented by data execution prevention (DEP) [16], a feature of all modern systems.

Requirements. The requirements that satisfy the goals of a lossless, scalable, and efficient hardware-enhanced CFI framework are given below.

- **Precision**: We must losslessly enforce any CFG with which we are provided. In general, it may be impossible to resolve a precise CFG either because source code is unavailable or the analysis is imprecise.
- **Scalability**: The effectiveness of any CFI approach depends on the CFG precision. Hence, we require that our CFI scheme scales to any level of CFG precision. Our solution must be capable of enforcing the CFI policy expressed by a given CFG.
- **Efficiency**: Software-based CFI approaches incur significant overhead, which limits adoption. We require negligible performance overhead for our CFI scheme.
- **Stateful**: We require stateful CFI since stateless CFI is vulnerable to stitching gadgets [14, 13] and control-flow bending attacks [6].
- **Compatibility**: Our CFI scheme needs to co-exist with uninstrumented programs.
- **Security**: Based on a precise CFG, we require the CFI scheme to cover all of the existing code-reuse attacks including traditional return-oriented programming [19], ROP without returns [8], just-in-time code-reuse attacks [21], and full function-reuse attacks [20].

3. HARDWARE-ENHANCED CFI DESIGN

We present a CFI approach that is able to losslessly enforce a program's CFG. As such, it must encode both coarse and precise CFG data and provide a mechanism to check that the encoded edges on the graph are properly traversed. At a minimum, our CFI policy ensures that forward edges (calls/jumps) target function entries and that backward edges (returns) target call preceded sites. Given a full CFG, our CFI policy ensures that forward edges must target their intended destinations. In our design, the precision of the protection of forward and backward edges is limited by the CFG coverage. In addition, we present a mechanism for uniquely identifying multiple indirect calls targeting a common function using a *trampoline*. For the purpose of encoding the CFG information, we introduce an extension to the ISA and a hardware module which enforces the CFG during execution, described below. The formal machine model for our CFI policy does not differ significantly from that presented by Abadi *et al.* in [3]. However, our design differs in that we are able to enforce any level of protection without incurring any extra overhead. Also, we are able to support different levels of protection as given by a CFG.

Inst.	**Semantics**
`cfibr  lbl`	Push `lbl` to top of LSS. Unique `lbl` issued per call.
`cfiret lbl`	Pop `lbl` from LSS and compare (returns only). Issued on valid return sites.
`cfiprj lbl`	Store `lbl` in LSR. Must precede indirect jump.
`cfiprc lbl`	Store `lbl` in LSR. Must precede call/tail call.
`cfichk lbl`	Compare `lbl` with value in LSR. Issued at indirect jump targets or function entries.

Table 1: Additions to the instruction set architecture

CFI Instruction Semantics and Instrumentation. We propose an instruction set architecture (ISA) extension as described in Table 1. This extension enables the dynamic creation of a stateful CFG and allow precise encoding, recording, and enforcement of a CFI policy. The execution-path behavior of the program is *encoded* in the ISA extension, where dedicated hardware is designed to check both the forward- and backward-edge state of the program. We validate forward- and backward-edge control-flow using CFI instructions, each of which encode a label that represents a valid/source destination pair for branching (call/jump/return) instructions.

Forward-edge control-flow is encoded by a CFI instruction, where the label (`lbl`) is a valid target determined by the CFG and written to the `label state register` (LSR). The label stored in the LSR is never written back to process memory, which at minimum prevents modification due to a memory corruption vulnerability and helps reduce performance overhead. Similar protection can be guaranteed by a SFI implementation, or via the x86 segment selector registers, where each label check is initiated by dereferencing memory, but would incur significant overhead due to the frequency of branching instructions. Backward-edge control-flow is encoded by the execution path's forward-edge behavior as a `cfibr lbl`, where the label `lbl` is written to the `label state stack` (LSS). In certain scenarios, it may be necessary to spill the LSS when overflown. We provide details for handling this condition below.

Addition of Trampolines. Multiple indirect calls may legitimately target a common function, which would force all associated `cfiprc`–`cfichk` pairs to have the same label. This condition results in an imprecise CFG which potentially allows an attacker to utilize unintended control flow

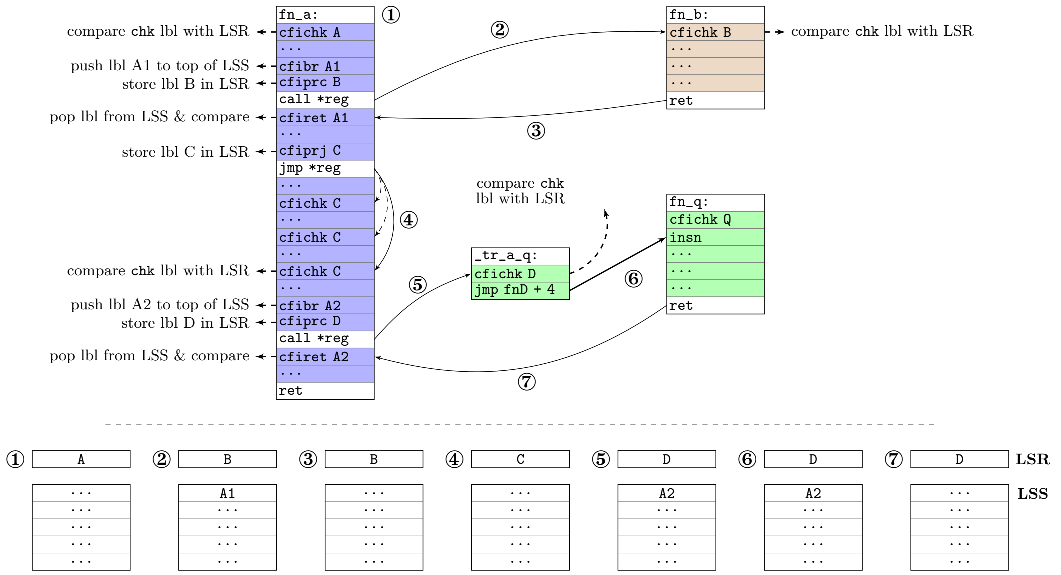

Figure 1: Tracking of a program's control flow graph. Matching numbers in the code segment match the LSS/LSR state.

paths in an exploit. We solve this by borrowing the concept of *trampolines* [25]. Trampolines are issued for every site that is targeted by multiple functions. Their addition transforms a call source/destination pair into a unique call source/trampoline pair. From the trampoline, a direct jump is issued to the original destination.

CFI Hardware Infrastructure. As mentioned above, a `label state stack` is used to record backward-edges to tightly couple caller/callee pairs and ensure only the most recently executed forward-edge is returned to. A `label state register` (LSS) is used to record forward-edges. This is used as some program semantics, such as case fallthrough, are inherently incompatible with a stack-based storage element. The CFI label data collected in the LSS and LSR is sensitive and therefore isolated from both the process' address space and the CPU's architectural registers. We enforce CFI through the ISA extensions using a state machine that supervises execution. The state transitions are encoded in the CFI instructions. If a violation of the CFI policy is detected a fault is triggered, resulting in the termination of the process. The LSS, LSR, and state machine control registers are mapped to physical memory in our implementation. The OS kernel maps this area and reserves it as part of MMIO. This allows the kernel to access and modify their contents.

Operating System Infrastructure. To support interoperability with commodity software not protected by our CFI implementation, we utilize the facilities of the Executable and Linkable Format (ELF) to request a custom loader from the system. This loader is modified to notify the kernel through the system call interface that the loaded process is requesting CFI protection. The process control block (PCB) of the kernel is extended to track which processes are instrumented. This allows us to support multiprocessing and shared libraries across unique CFI protected programs. Upon task switch, the OS enables/disables the CFI hardware and backs-up/restores the state machine registers, LSS and LSR for the process using the MMIO interface described above.

If a CFI enabled process creates a clone of itself, or a thread, the OS also clones the current CFI context in the clone's PCB. If the forked process then executes the `execve` system call, its CFI entry is cleared in the PCB and CFI protection is disabled for the newly loaded image. CFI protection can then be requested by the new process. Process termination is handled as normal, deallocating the PCB for the process from kernel space and performing the standard cleanup tasks.

4. DETAILED OPERATION

We will use Figure 1 to highlight the operation of our CFI platform. The top portion of the figure shows an instrumented code snippet, while the bottom portion shows the different states of the LSS and LSR during execution.

Execution begins at ①, where the label `A` stored in the LSR is verified against the `cfichk A` instruction at the function entry. In our implementation, this check is an XOR operation using the CFI instruction's immediate value and the value stored in the LSR as operands. Prior to executing the call at ②, a `cfibr A1` instruction pushes the label `A1` to the LSS and the `cfiprc B` instruction updates the entry in the LSR with `B`. Pushing `A1` to the LSS preserves the backward-edge state of the call and creates a unique return site for every executed call in the process. Even in the case where this call targets multiple functions, we maintain a unique call/return pair because the return policy is separated from the call policy, the `cfibr lbl` for a return and `cfiprc lbl` for calls. This allows us to prevent an adversary from targeting any return site, an attack technique employed against coarse CFI policies.

Once the call instruction is executed, the `cfichk B` instruction at the target's entry is compared with the label `B` stored in the LSR, validating the call target. Upon return at ③, the return instruction targets the `cfiret A1` instruction, which pops the last stored label (`A1`) from the LSS and performs a comparison. The `jmp` instruction at ④ is preceded with a `cfiprj C` instruction. This stores the label `C` in the LSR much like the `cfiprc lbl` prior to a function call. After the `jmp` instruction is executed, a check state is entered, wherein a CFI comparison is performed with a `cfichk C` instruction before execution is allowed to continue. An additional feature of our hardware-assisted CFI approach is that it enforces the exact execution of the aforementioned sequence of instructions, ensuring an attacker is unable to bypass the CFI checks.

Ideally, all indirect call sites target unique sets of functions, but this is not guaranteed. If two indirect call sites share a target, then the labels in the preceding `cfiprc` instructions must be the same. This introduces coarseness into the CFI policy that can be leveraged by an adversary by redirecting control flow along erroneous, but valid, control flow edges. We borrow trampolines [25] as a technique to avoid this behavior. Trampolines act as a springboard from the caller to its intended callee. The indirect call made at ⑤ illustrates this technique. In our example, `fn_q` is targeted by multiple callers. A trampoline is inserted from `fn_a` that is unique to the call site along ⑤. In this case, the indirect call initiates a check wherein the `cfichk D` instruction is compared with the current label in the LSR (`D`). Once the check is verified, a direct jump is made to the second instruction in the target function along ⑥ to avoid an invalid state transition. A `cfichk Q` instruction remains at the entry to `fn_q` because one function among all of its callers does not have to target a trampoline. The return along ⑦ is executed in the same manner as ③.

5. PERFORMANCE EVALUATION

To evaluate the support of our hardware-enhanced CFI protection we generated custom build tools, runtime environment, and hardware infrastructure. This enabled us to (i) issue newly added CFI instructions in proper code locations, (ii) create unique CFI labels for any arbitrary application, and (iii) support CFI services within a rich-OS environment on a hardware platform.

Build tools. We developed an instrumented toolchain based on the GNU compiler Collection (`gcc`) version 4.9.2, the GNU Binary Utilities (`binutils`) version 2.23 and μClibc (`uClibc`) version 0.9.33.2. Routines written in assembly were manually instrumented.

Hardware Platform. Our design was integrated with the open-source LEON3 processor distributed by the European Space Research and Technology Centre [18]. The LEON3 is a 32-bit processor that implements the SPARC V8 ISA [1], and is equipped with a 7-stage pipeline, separate instruction and data caches, memory management unit, hardware floating-point units, AMBA 2.0 AHB bus, and on-chip debug support. Modifications were made parallel to the processor pipeline write-back stage to incorporate the CFI-FSM, LSR, and LSS in the `iu3.vhd` module. We also made minor additions to the decode stage so that the CFI instructions are decoded as `nop` instructions, which ensures single cycle latency as determined by the SPARC V8 ISA [1]. A Xilinx KC705 evaluation board was used as our test platform.

Hardware-Enhanced CFI Performance Results. We used the industry standard EEMBC's CoreMark benchmark suite [2] for our performance evaluation. This benchmark suite is designed to test a processor core's functionality, namely its pipeline, memory access, and functional unit operations. The test suite covers usage of code pointers and provides frequent conditional/unconditional branching, which provides a representative class of CFI instrumented code coverage for comparison.

We evaluated several SPEC INT2006 benchmarks, namely `bzip2`, `libquantum`, and `h264ref`. These are representative example programs from the group of business, scientific, and problem-solving workloads. We did not evaluate full SPEC because of resource constraints on the FPGA evaluation board. The FPGA board provides 1GiB of main memory, whereas full SPEC requires at least 1GiB of free memory. Each of the programs evaluated could be run within the memory constraints imposed by the FPGA platform. The benchmarks evaluated offer a reasonable tradeoff in build-time and coverage.

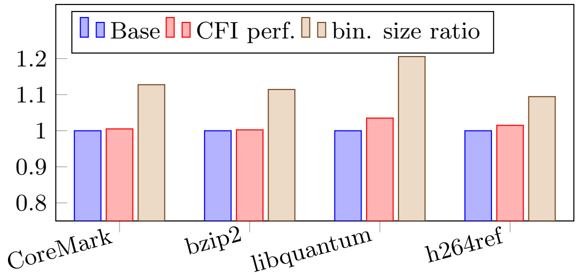

Figure 2: Normalized benchmark results.

For testing and CFG instrumentation purposes, an IDA Pro plugin that extends the SPARC processor module bundled with the program was written. This enabled us to automatically instrument backward edges in our binary. Forward edges for indirect jumps were instrumented by manually extracting jump table information from the binary and feeding this information to the plugin. Indirect calls were instrumented to match source-target labels. For testing purposes, we did not instrument trampolines, which relaxed the control flow graph. We do not believe this significantly affects our performance results, since the ratio of indirect calls to total calls for SPEC INT2006 averages 4.1%. Furthermore, most of these are found within the runtime environment. The main computational load of the SPEC benchmarks are in the program itself. Also, as the overhead for our implementation is mainly due to code size, the full instrumentation of shared objects will not affect performance.

The results are shown in Figure 2, where the performance overhead on average is 1.75%, with a worst-case overhead of 3.5% for SPEC benchmarks and 0.5% for CoreMark. The average code size overhead is 13.5% across both SPEC and CoreMark. We should note that this overhead is directly related to the number of calls and indirect jumps in the binaries.

	LEON3	**LEON3-CFI**	**% Change**
`comb.`	8759.073	8996.952	2.72
`seq.`	16921.416	17143.284	1.31
`total`	25680.589	26140.236	1.78

Table 2: Evaluation of area overhead with CFI implemented on a LEON3 processor.

Area and Timing Overhead. Our hardware-enhanced CFI LEON3 core was synthesized with Design Compiler H-2013.03-SP5-3 using the Synopsys 32/28 nm generic library, a teaching library created for microelectronic design education. We evaluated both area and maximum clock rate. In general, smaller area ensures better resource usage and lower cost requirements. A faster clock ensures our hardware will not be on the critical path or violate existing timing constraints and stall the pipeline. Table 2 displays the area overhead caused by extending the pipeline with full CFI protection. The total area overhead is a negligible 1.78%.

We also evaluated the maximum frequency at which our

CFI-FSM, LSS, and LSR implementation could be clocked as a stand alone module and determined using Design Compiler timing scripts that our additions could be clocked up to 3 GHz without incurring timing violations.

6. SECURITY EVALUATION

The main goal of our hardware-enhanced CFI platform is to prevent code-reuse attacks that leverage either invalid backward edges, forward edges, or full functions. These include attacks that corrupt return addresses [19], code pointers used in indirect calls/jumps [8, 7], or reuse entire functions [20, 20, 23]. Finally, we must prevent runtime attacks that bypass CFI while adhering to its policies [6].

For our security discussion, we consider the adversary model and assumptions mentioned in Section 2. Due to page constraints we can not include a full example of a CRA, however, we do evaluate a prototype exploit and refer the interested reader to [19, 8, 7, 20, 23] for details. We assume the most precise CFG has been given.

Backward-Edge Code Reuse attacks. We prevent backward edge runtime attacks described here, and in general, because they require redirection to invalid call-preceded instructions or arbitrary code locations. This is in direct violation of our method of ensuring precise state preservation. Each call instruction is instrumented with a unique label that encodes the execution path's state information with a `cfibr lbl/cfiret lbl` instruction pair. A return instruction is only allowed to target a `cfiret` instruction if it is the most recent in the execution path history, i.e., it is a valid state. This is determined by checking the label at the top of the LSS against `cfiret lbl` at the return target. Only `cfiret` instructions may be targeted by returns.

Forward-Edge Code Reuse Attacks. We prevent forward edge runtime attacks described above, and in general, because their reliance on either an arbitrary location or to an invalid call/jump target. Only valid indirect call/jump targets are allowed as given by a program's CFG. This prevents the attacker from redirecting control-flow to arbitrary locations in program code. Each benign call/jump target is instrumented with a `cfipr* lbl/cfichk lbl` pair that encodes its intended targets. Redirection to an invalid target is prevented when the check against the stored label in the LSR fails. Redirection to an arbitrary location in the application's code space will not target `cfichk` instructions.

While we did not port our instruction extensions to a JIT compiler in this work, our hardware-enhanced CFI architecture can support protection against dynamic code-reuse attacks if a JIT compiler were modified. Dynamic CRAs, such as JIT-ROP, dynamically determine gadgets on executable memory pages [21]. These attacks exploit backward and forward edges in the same way we have described herein and are therefore prevented by our platform.

Full-Function Code-Reuse Attacks. Our hardware enhanced CFI prevents full-function reuse attacks, such as COOP [20], because they rely on redirecting control-flow to an invalid indirect call/jump targets. Valid branch targets are instrumented with `cfipr* lbl/cfichk lbl` pairs. If the class hierarchy is correctly and precisely covered in the CFG, then only benign control-flow targets are encoded with matching labels. For instance, Google compiler extensions can be leveraged to extract such precise CFG information [22]. Redirection to an invalid control-flow target is prevented by checking that the label currently in the LSR matches the `cfichk lbl` label.

Control-Flow Bending. A recent attack called control-flow bending (CFB) demonstrates code-reuse attacks are possible while adhering to fully precise static CFI [6]. In a CFB attack, an attacker may corrupt a code pointer to call a valid function entry, where a vulnerability exists allowing to corrupt a return address. The corrupted return address may then be used to return to a call-preceded site. CFB exploits any function with a vulnerability that can overwrite its own return address and adheres to CFI by returning to any location where this function was called.

We prevent CFB attacks since they require redirection to any call-preceded slot in a stateless CFI protection system. We offer precise, *stateful* CFI so that only the most recently executed forward-edge transition may be returned to. As described above, this is ensured with a unique `cfibr lbl/cfiret lbl` instruction pair. A return instruction is only allowed to target a call-preceded slot if it is the most recent in the execution path history.

Security of Label State Stack/Register. Even though it is not a strict security requirement, our design only allows CFI instructions to access the LSS and LSR from userspace and avoids CFI data being loaded to main memory. Recall the recent CFI attack that corrupts offset pointers referencing a CFI jump table spilled to the program's stack due to efficiency reasons [10]. We prevent this attack, and similar attacks that corrupt or disclose CFI data, by storing CFI-related data such as labels in a dedicated memory, the LSS and LSR, and backing them into kernel memory when necessary.

Sample Exploit. As a baseline test of our protection, we developed a program that exploits a buffer overflow in the stack, thus overwriting the return address of a function. We pointed this address to a call-preceded site to implement a loop, attempting to adhere to a coarse-grained CFI policy. Our system was capable of detecting this violation. We also redirected the return to arbitrary return sites with equal results. Our exploit also allowed us to change the target of an indirect call in the program's code to an arbitrary location. The violation on control flow was detected whenever an invalid location, function or otherwise, was targeted.

7. RELATED WORKS

In the original CFI work, Abadi et al. [3] propose a label-based mechanism using a software based implementation. Unfortunately, software-based instrumentation induces too high performance penalties. A number of coarse-grained CFI approaches aim at tackling the performance overhead with some solutions building on heuristics or relaxing the CFI scheme [17, 9, 24]. However, a number of recent attacks against CFI demonstrates that such policies are bypassable [6, 20]. In contrast, our hardware-based CFI scheme allows for fine-grained policies that resist these attacks while still being highly efficient.

Architectural fine-grained CFI support, as proposed by [5], introduced hardware support for fine-grained CFI protection via integrity checking of control-flow graph (CFG) encoding. For forward-edge protection, Budiu et al. [5] leverage a CFI label register similar to our LSR. However, for backward-edge protection, they assume a shadow stack which incurs more performance overhead compared to our LSS. Similarly, Davi et al. [12] introduce hardware-assisted CFI instructions but only focus on CFI backward edges, and bare metal code.

In contrast, we support highly efficient CFI for shared libraries, multitasking, and support of legacy code.

8. CONCLUSION AND FUTURE WORK

Within this paper we present the design and implementation of a lossless, scalable and highly efficient hardware-enhanced CFI platform. The new framework leverages dedicated CFI instructions to losslessly enforce any CFG and diverse CFI policies within our model. Our hardware-enhanced CFI significantly lowers the performance overhead when applied to several SPEC INT2006 and CoreMark benchmarks. Further, if provided with a precise CFG we show comprehensive protection from many traditional and recently proposed code-reuse attacks. The goal of our work is the design and implementation of a hardware-enhanced CFI framework that can losslessly support CFI policies with varying precision. Our future work will investigate both CFG generation and extend support to additional instruction set architectures (ISAs).

9. ACKNOWLEDGEMENTS

We would like to extend our gratitude to Christopher Liebchen for his input at the start of this project. This work has been co-funded by the German Science Foundation as part of project S2 within the CRC 1119 CROSSING and the European Union's Seventh Framework Programme under grant agreement No. 609611, PRACTICE project.

10. REFERENCES

[1] SPARC International Inc. SPARC V8 processor. http://www.sparc.org.

[2] The embedded microprocessor benchmark consortium: EEMBC benchmark suite. `http://www.eembc.org`.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS'05, 2005.

[4] W. Arthur, S. Madeka, R. Das, and T. Austin. Locking down insecure indirection with hardware-based control-data isolation. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 115–127. ACM, 2015.

[5] 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*, ASID'06, pages 42–51, 2006.

[6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[7] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[8] 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*, CCS'10, 2010.

[9] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS'14, 2014.

[10] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963. ACM, 2015.

[11] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566. ACM, 2015.

[12] 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.

[13] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[14] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, SP'14, 2014.

[15] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu Ghazaleh. Signature-based protection from code reuse attacks. *Computers, IEEE Transactions on*, 64(2):533–546, 2015.

[16] Microsoft. Data execution prevention (DEP), 2006.

[17] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[18] G. Research. LEON3 synthesizable processor.

[19] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, 2012.

[20] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, SP'15, 2015. To appear.

[21] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, 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*, SP'13, 2013. Received the Best Student Paper Award.

[22] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[23] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, 2011.

[24] F. Yao, J. Chen, and G. Venkataramani. Jop-alarm: Detecting jump-oriented programming-based anomalies in applications. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 467–470. IEEE, 2013.

[25] 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. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, SP'09, 2009.

[26] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, SP'13, 2013.