HAFIX: Hardware-Assisted Flow Integrity Extension

Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany

Patrick Koeberl
Intel Labs

Dean Sullivan, Orlando Arias, Yier Jin
University of Central Florida, USA

ABSTRACT

Code-reuse attacks like return-oriented programming (ROP) pose a severe threat to modern software on diverse processor architectures. Designing practical and secure defenses against code-reuse attacks is highly challenging and currently subject to intense research. However, no secure and practical system-level solutions exist so far, since a large number of proposed defenses have been successfully bypassed. To tackle this attack, we present HAFIX (Hardware-Assisted Flow Integrity eXtension), a defense against code-reuse attacks exploiting backward edges (returns). HAFIX provides fine-grained and practical protection, and serves as an enabling technology for future control-flow integrity instantiations. This paper presents the implementation and evaluation of HAFIX for the Intel® Siskiyou Peak and SPARC embedded system architectures, and demonstrates its security and efficiency in code-reuse protection while incurring only 2% performance overhead.

1. INTRODUCTION

Code-reuse attacks have become the state-of-the-art technique to exploit memory-related vulnerabilities in modern software. These attacks require no code injection. That is, they hijack the intended control-flow of applications and redirect it to unintended but valid code sequences. Hence, these attacks circumvent non-executable memory protection against code injection attacks, which is currently deployed on many computing platforms including Intel®, ARM, and SPARC.

In particular, code-reuse attacks based on return-oriented programming (ROP) combine short code sequences (called gadgets) residing in shared libraries to induce malicious program actions [15, 5]. These code sequences typically terminate in a return or indirect jump/call instruction. ROP attacks have been launched on various architectures such as Intel® x86, ARM, SPARC, PowerPC, and embedded systems (e.g., Atmel AVR sensors [9]). Indeed, many of today's exploits are based on ROP techniques. This fact has made the design and implementation of practical ROP mitigation a hot topic of research. The most general solution against ROP is *control-flow integrity* (CFI) as it restricts program execution to a pre-defined control-flow graph (CFG) [3]. Unfortunately, the original CFI proposal [3] suffers from practical deficiencies, most notably performance overhead. Hence, recent CFI approaches aim at a tradeoff between security and practicality [18, 13]. In particular, they relax the more restrictive policies presented in [3] to gain efficiency. However, these so-called coarse-grained CFI policies can be undermined as shown recently [11, 8]. Until now, the majority of research on CFI has focused on software-based solutions.

Our goal and contributions.

We take a hardware-based CFI approach that has several advantages over software-based counterparts: First, it is significantly more efficient, as we demonstrate. Second, compiler support is simplified by reducing complex CFI checking code to single CFI instructions. Third, dedicated CFI hardware instructions and separate CFI memory provide strong protection of critical CFI control-flow data.

We focus on backward-edge CFI, that is, CFI for indirect branches through function return instructions in the program's CFG. In contrast, forward-edge CFI handles indirect branches in the CFG that are caused by jumps and function calls [16]. Forward-edge CFI can be efficiently enforced by software solutions [16] whereas conventional ROP and many public real-world code-reuse attacks exploit backward edges. In terms of performance, protecting backward edges is more challenging, simply due to the fact that function returns occur far more frequently than indirect calls and jumps. Existing backward-edge CFI schemes either suffer from large performance degradation (when using a shadow stack for return addresses), or they deploy vulnerable, coarse-grained CFI policies as mentioned above.

Our contributions are as follows: We developed and evaluated for the first time a fine-grained backward-edge CFI system in hardware, called HAFIX. We adapt the conceptual work by Davi et al. [7] and extend existing processors' instruction set architectures.

To develop HAFIX, we had to tackle several challenges: (i) efficient and secure access to CFI control-flow data (i.e., label memory), (ii) specifying new CFI instructions that perform efficiently in one cycle, and (iii) automatically emitting these instructions during compilation. Additionally, we also tackled the challenge of providing CFI protection for recursive function calls in our architecture.

We present real hardware implementations of backward-

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA
Copyright © 2015 ACM 978-1-4503-3520-1/15/06 ...$15.00.
http://dx.doi.org/10.1145/2744769.2744847.

edge CFI targeting bare metal code. Specifically, we extend the processors' instruction set with new CFI instructions, and also modify the respective compilers to automatically emit our new instructions into applications. We show proof-of-concept implementations on two different processor architectures: The first is the Intel® Siskiyou Peak platform which primarily targets embedded applications [14]. The second is the open source LEON3 microprocessor which implements the SPARC V8 instruction set and has been developed by the European Space Agency for avionic applications [1].

One significant aspect of our work is the performance and security evaluation of fine-grained hardware-based (backward-edge) CFI that we conducted on embedded systems under different security scales. We evaluated our HAFIX implementation using standard embedded benchmarks (including Dhrystone and CoreMark), and show that it only adds 2% of performance overhead on average. Moreover, we provide a security evaluation to demonstrate that HAFIX reduces the available gadget space to 19.82% on average compared to recently proposed defenses in a worst-case setup (cf. Section 5.3).

2. SYSTEM MODEL

In this section, we present our target threat model, main assumptions, and requirements.

Threat Model. Our main goal is to thwart code-reuse attacks launched through backward edges (function returns). The adversary i) has full control over the program's stack and heap to inject new return addresses and overwrite existing ones, ii) has access to the application's code including linked libraries, iii) can exploit a buffer error to instantiate a code-reuse attack, and iv) can bypass any deployed code randomization (e.g., ASLR), i.e., the adversary has full knowledge of the application's memory layout.

Assumptions. As the main scope of this paper are CFG backward edges, we assume that the target system deploys software-based CFI protection for forward edges [16] and leave hardware-assisted support for these edges for future work. We also assume that the target hardware platform enforces protection against code injection (e.g., NX-Bit) currently deployed by default on almost all platforms.

Requirements. The main objectives of our CFI solution are efficiency, practicality, and enforcing fine-grained CFI protection on backward edges. Fulfilling these requirements is highly challenging, since function returns occur frequently at runtime. Recent research has shown that coarse-grained CFI policies for returns, i.e., restricting returns to target a call-preceded instruction, are insufficient and can be bypassed [11, 8].

3. DESIGN

Our hardware-assisted CFI solution HAFIX adapts the concept proposed by Davi et al. in [7]. Moreover, we tackle the problem of recursive function calls. Despite proper design, we aim at providing a practical implementation of fine-grained CFI as well as a comprehensive evaluation of our solution on real embedded systems hardware.

3.1 Backward-Edge CFI Scheme

Fine-grained software-based CFI approaches validate function returns based on the so-called *shadow stack* (or return address stack) paradigm (e.g., [3]): all return addresses that are pushed on the program's stack through call instructions are tracked on a separate, protected shadow stack. Upon function return, CFI verifies whether the program uses a return address that is held on the shadow stack. If not, control-flow is violated; otherwise the program will continue execution and the used return address is popped off the shadow stack. Although shadow stacks provide fine-grained protection, they (i) significantly decrease performance and (ii) lead to false positives for certain programming constructs (C++ exceptions with stack unwinding, setjmp/longjmp). Concurrently to our work, Dang et al. [6] demonstrate that performance can be increased by leveraging a parallel shadow stack. However, the parallel stack still resides in the same address space of the target application.

In our solution HAFIX, we adopt the idea of confining function returns to active call sites [7]. In other words, we force a return to target a call-preceded instruction inside a function that is currently executing. As we will show, this CFI policy can be efficiently implemented in hardware and requires only minimal changes to the compiler.

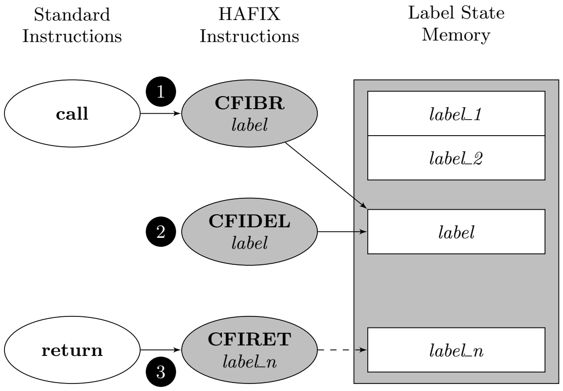

Figure 1: Abstract design of HAFIX

The underlying design to enforce this CFI policy is depicted in Figure 1. In order to monitor functions that are currently executing, HAFIX requires the compiler to assign unique labels to each function. Further, it forces the first instruction of each function to be a `CFIBR`. This instruction loads the label of the function into a dedicated memory area, called the *label state memory*, to indicate that the function is active (Step ❶). Internally, direct and indirect call instructions lead to a processor state switch in which the processor only accepts `CFIBR`. To deactivate a function, HAFIX uses the `CFIDEL` instruction which effectively removes the label from the label state memory (Step ❷). Hence, `CFIDEL` instructions are executed just before a function return instruction.

The critical point of backward-edge CFI is the final function return instruction of the subroutine, since this indirect branch instruction can be exploited by the adversary to hijack the program's control flow based on a malicious return address. However, in HAFIX, return instructions need to target an active call site. To enforce this, only returns to the `CFIRET` instruction are permitted, in particular those `CFIRET` instructions that define a currently active label in the label state memory (Step ❸). The dashed line in Figure 1 indicates that `CFIRET` does not change the label state, but only checks whether a label is active. We will give a concrete code example in Section 4.2.

Implementation Requirements. We need to define new hardware instructions CFIBR, CFIDEL, and CFIRET. Further, we need to implement a state model that switches states on function call and returns to only accept as next instructions CFIBR and CFIRET respectively. On the compiler side, we need to emit these instructions at their corresponding places: CFIBR at function start, CFIRET at all call sites, and CFIDEL at function return.

3.2 Recursive Function Calls

Our design and implementation needs to tackle an important challenge of handling recursive function calls. These lead to a number of store operations of the same label, since a recursive function invokes itself several times before each instance returns. To solve this problem, we only store the label once and record the number of invocations in a separate (hidden) register. For this, we introduce a new CFI instruction called CFIREC and a new shadow register called CFIREC_CNTR.

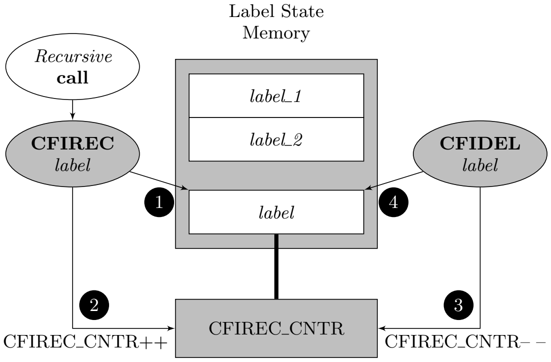

Figure 2: Recursion handling in HAFIX

The abstract workflow of a recursive call under our new CFI instrumentation is shown in Figure 2. First, the compiler emits CFIREC at the beginning of all recursive functions (rather than a standard CFIBR). Upon execution, CFIREC activates the label of the recursive function but only if the CFIREC_CNTR is set to zero (Step ❶). In addition, CFIREC increments the CFIREC_CNTR (Step ❷). Internally, we also associate the label of the recursive function to the CFIREC_CNTR register.

Upon function return, the CFIDEL instruction validates whether the current label is associated to CFIREC_CNTR. If the link exists, HAFIX knows that a recursive function is active and we subsequently check whether CFIREC_CNTR > 1. If so, we only decrement CFIREC_CNTR (Step ❸) as more returns from the same function are expected. The label is only removed from memory if CFIREC_CNTR is equal to 1 (Step ❹), indicating the last function instance of the recursive function returns. Note that our mechanism targets non-nested recursive functions. Nested recursive functions are rare in embedded applications, e.g., none of our benchmarks contained them (see Section 5).

4. IMPLEMENTATION OF HAFIX

In this section, we present the implementation of HAFIX on our target architectures Intel® Siskiyou Peak and SPARC. We give a short overview on both architectures, present HAFIX-instrumented code, and finally describe the implementation of the CFI instructions and the label memory in hardware.

Siskiyou Peak

```
<funct_a>
 1: cfibr 0x15
 2: push %ebp
 3: mov %ebp, %esp
 4: call <funct_b>
 5: cfiret 0x15
 6: mov %esp, %ebp
 7: pop %ebp
 8: cfidel 0x15
 9: ret
```

```
<funct_b>
 a: cfibr 0x16
 b: push %ebp
 c: mov %ebp, %esp
 d: asm_ins...
 e: mov %esp %ebp
 f: pop %ebp
10: cfidel 0x16
11: ret
```

SPARC

```
<funct_a>
 1: cfibr 0x15
 2: save %sp, -96, %sp
 3: sparc_ins ...
 4: call <funct_b>
 5: nop
 6: cfiret 0x15
 7: restore
 8: retl
 9: cfidel 0x15
```

```
<funct_b>
 a: cfibr 0x16
 b: save %sp, -96, %sp
 c: sparc_ins ...
 d: restore
 e: retl
 f: cfidel 0x16
```

Figure 3: HAFIX-instrumented code

4.1 Intel® Siskiyou Peak and SPARC

Intel® Siskiyou Peak is a 32-bit, fully synthesizable core intended for deeply embedded applications [14]. The core is highly configurable and features a 5-stage, single-issue processor pipeline. Major configuration options include a Memory Protection Unit, various branch predictors, I&D caches and multiplier performance options. Siskiyou Peak also includes a variety of micro-architectural options to trade off clock frequency for improved instructions-per-cycle. The processor is organized as a Harvard architecture with separate busses for instruction, data and memory mapped IO spaces. The instruction set is a small subset of the 32-bit x86 instruction set and shares the same variable-length binary encoding.

The SPARC implementation of HAFIX was based on the open-source LEON3 processor. Initially developed by the European Space Research and Technology Centre, the Aeroflex Gaisler LEON3 is a synthesizable 32-bit processor containing a 7-stage pipeline with Harvard architecture, memory management unit, hardware multiplier and divider, on-chip debug support and multi-processor extensions. The LEON 3 implements the SPARC V8 architecture [1], a 32-bit architecture which provides 3 to 32 register windows, where each window offers 32 general purpose registers. The core is extendable by means of its AMBA 2.0 AHB bus interface supporting IP core plug and play [10].

4.2 Code Instrumentation

Figure 3 shows HAFIX-instrumented assembler code targeting Intel® Siskiyou Peak and SPARC. The example shows two sample functions with their function prologue and epilogue instructions, where funct_A simply calls funct_B. The emitted CFI instructions are highlighted with blue color.

In both architectures, our instrumented compilers prepend CFIBR to function prologues that activate the unique label of the function in the label state memory. Further, CFI-

DEL is appended to function epilogues to mark the end of a function resulting in a label de-activation (see also Figure 1). On SPARC, we replace the `nop` instruction which is normally found after the `retl` instruction. Hence, emitting `CFIDEL` comes at no extra cost in terms of space and performance. Lastly, `CFIRET` is inserted after a `call` instruction. Due to the design of the SPARC pipeline, in this architecture we emit `CFIRET` after a pipeline bubble (`nop`) commonly found after a `call` instruction.

Unique labels are added to these instructions within their context with a program that is executed after the linker. On SPARC, we modified the Aeroflex Gaisler Baremetal C Compiler (BCC) and accompanying libraries to insert our new CFI instructions into programs [2]. These are based on GNU Binutils and GCC. Instrumenting the assembler was done through modification of `libopcode`, whilst the output templates and code generation in GCC were modified to accommodate the required behavior. For Siskiyou Peak, we modified the customized LLVM compiler toolchain (shipped already with Siskiyou Peak) to generate HAFIX-instrumented code. Moreover, during a post-compilation processing phase, we identify recursive function calls and replace the corresponding `CFIBR` with `CFIREC` instructions.

4.3 Hardware CFI and Label Memory

Siskiyou Peak. Hardware CFI enforcement on Siskiyou Peak is achieved by augmenting the execution stage of the processor pipeline with a CFI control unit and associated label state memory, see Figure 4. The CFI control unit monitors CFI instruction sequencing and manages CFI label state. In the event of a CFI violation being detected, a processor exception is issued allowing a software handler to take appropriate action.

Label state memory is implemented as a tightly-coupled 16384x1 memory with the CFI label employed as the index. This facilitates highly efficient CFI instructions: For a given label, `CFIBR` sets the memory location indexed by the label while a `CFIDEL` clears it. The `CFIRET` instruction reads the location indexed by the label and raises an exception if not set. In the target platform of Xilinx Spartan-6 this approach allows CFI label state to be efficiently mapped onto two synchronous Block RAMs. Due to the low logic complexity of the indexing mechanism it is feasible to clock the Block RAM on the opposite clock edge removing a cycle of read latency and enabling single-cycle performance for all CFI instructions.

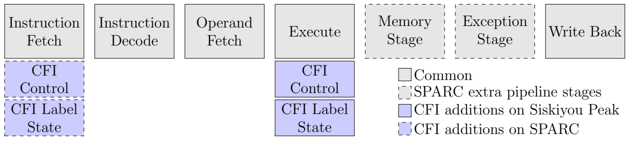

Figure 4: CFI pipeline integration

SPARC. Our implementation of HAFIX in SPARC hardware is based around the Aeroflex Gaisler LEON3 microprocessor, which provides a fully compliant SPARC V8 core. To enforce a control-flow mechanism, a finite state machine (FSM) was developed and our new CFI instructions were added to the SPARC V8 Instruction Set. The FSM is connected into the processor pipeline at the fetch stage, using the incoming instructions as inputs. Because of the 7-stage pipeline the LEON3 core provides, extra states were added to the FSM to account for pipeline bubbles, stalls and flushes.

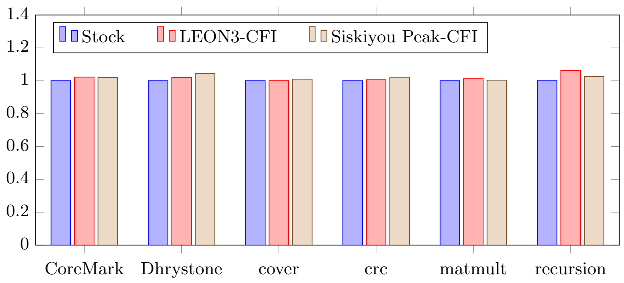

Figure 5: CFI extension overhead w.r.t stock core for LEON3 and Siskiyou Peak

The HAFIX-FSM was built in the 7-stage integer pipeline in parallel to the fetch unit. As per the design requirement in Section 3.1, the FSM regulates instruction execution in order enforce secure control flow via reading and writing to the HAFIX label memory based on issued HAFIX instructions.

CFI label memory was designed as a 1024x13 last-in-first-out (LIFO) structure to strictly enforce call/return pairs. As shown in Figure 4, the CFI control unit and label memory are implemented separate from the CPU bus, so that software has no access to its internal state. Upon encountering a `CFIBR` instruction, the last label stored in the LIFO structure is checked, if different from the label on the `CFIBR` instruction, the new label is pushed in, otherwise, a counter associated with the label is incremented. On `CFIDEL`, the counter associated with the last label is checked, if non-zero, it is decreased by one, if zero, the label is popped from the LIFO (see also Section 3.2). This handles recursive calls without exhausting the storage capabilities of the LIFO. On `CFIRET`, the last label pushed to the LIFO is checked, if different from the `CFIRET`, a fault is triggered, halting the CPU, ensuring proper control flow, otherwise, execution continues as normal.

5. EVALUATION

5.1 Performance

The system impact was evaluated using a suite of microprocessor benchmarks including CoreMark, Dhrystone, cover, crc, matmult and recursion. The performance overhead for the HAFIX-enhanced Siskiyou Peak and LEON3 cores is shown in Figure 5 with the respective unmodified stock core used as the baseline. The overall performance overhead is around 2% for both architectures with backward-edge CFI enabled. The largest increases, as expected, are seen in those benchmarks that include many short function calls such as Dhrystone and recursion. None of our benchmarks programs raised a false CFI violation. As discussed by Dang et al. [6] several shadow stack implementations require special handling of certain programming constructs to avoid a false alarm.

5.2 Area

The performance of the implemented architecture with respect to area was evaluated using results from the Xilinx place and route (PAR) tools. The HAFIX enhanced Siskiyou Peak core consumes an additional 2.49% registers and less

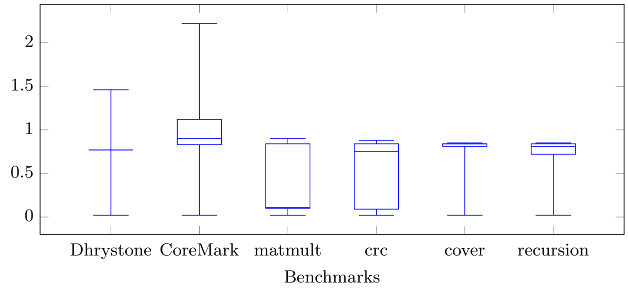

Figure 6: Percentage of program instructions that a function return is allowed to target

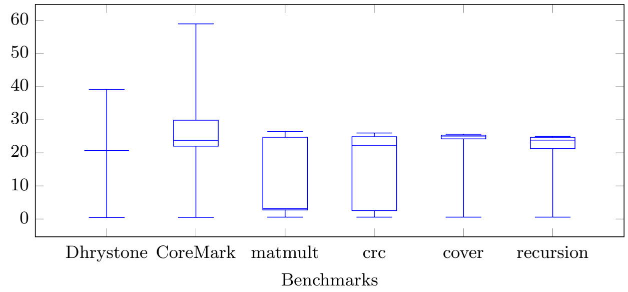

Figure 7: Percentage of CFIRET instructions that a function return is allowed to target

than 1% additional LUTs (Look Up Tables). The CFI label state memory is implemented using 2 Block RAMs.

For the SPARC implementation, the area was similarly evaluated using results from Xilinx PAR tools. The HAFIX enhanced LEON3 core consumes an additional 2.97% registers and 0.33% additional LUTs. The LIFO implementation incurred an 8% increase in distributed RAM usage.

5.3 Security

To measure to what extent HAFIX reduces the set of valid branch addresses, we record the label memory at each function return for all of our benchmark programs under the CFI implementation for Siskiyou Peak. The chart shown in Figure 6 demonstrates that on average only 0.70% of all program instructions are addressable by a function return; with a maximum of 2.2% (CoreMark).

In order to give our evaluation more meaning, we directly compare our backward-edge CFI realization to recent CFI-based approaches that restrict returns to target a call-preceded instruction [13, 18]. For this, we validate how many CFIRET instructions (i.e., call-preceded instructions) a function can target on average. Figure 7 shows that the median percentage of valid CFIRET instructions for the individual benchmarks ranges from 3.13% (matmult) to 25.36% (cover). Hence, HAFIX significantly reduces the gadget space (to 19.82% on average) compared to recent CFI-based approaches.

Note that using static-linked benchmark programs for our security evaluation resembles a worst-case scenario. In fact, all the numbers reported would be tremendously lower for dynamically-linked programs for two reasons. First, shared libraries introduce a large amount of code that is never used during program execution. In coarse-grained CFI, all call sites inside the shared library are valid targets, but in HAFIX only those call sites of the invoked shared library function are valid. Second, benchmark programs typically contain a large (main) function that invokes a number of subroutines. As the main function remains active almost throughout the entire program execution, all its call sites (i.e., CFIRET instructions) are valid targets for function returns. As an example, Dhrystone contains a large main function with 87 call sites out of 419 call sites in total. However, even in this circumstance, HAFIX still reduces the gadget space to 20% compared to recent CFI-based approaches. In order to further reduce the space, we can emit multiple labels into the main function, e.g., splitting the Dhrystone main function into four parts reduces the set of addressable CFIRET instructions to 5%.

Note that an adversary cannot undermine HAFIX by modifying the CFI label state or the CFIREC_CNTR register simply due the fact that both are not directly accessible by software, but only by our CFI instructions. Since all modern platforms as well as our target architectures prevent code injection attacks (using data execution prevention), an adversary can neither modify the CFI-protected code nor inject malicious CFI instructions.

Practical Exploits: We also evaluated the effectiveness of HAFIX using code-reuse exploits against self-developed vulnerable programs. Our attack on SPARC is initiated by overflowing a buffer, which results in the eventual overwrite of the register holding the return address, %i7. Similar to a conventional return-into-libc attack, our malicious return address points to the start of a payload. However, upon returning from the function, HAFIX reports a control-flow violation since the exploit jumps to an address that does not match a valid CFIRET site. Similarly, on Siskiyou Peak our ROP exploit returns to an invalid CFIRET site. Once the HAFIX invalidation occurs, a CPU reset trap terminates code execution in the exception detection stage.

6. RELATED WORK

Over the last decade many defenses have been proposed to mitigate runtime exploits. Due to the page limit, we focus only on closely related hardware-based/assisted CFI solutions.

The hardware-based CFI state model was first proposed by Budiu et al. [4]. The main idea is to directly embed unique labels in indirect branch instructions and emit cfilabel instructions at possible indirect branch targets. When an indirect branch executes, it loads its encoded label into a dedicated register. Afterwards, the state model forces the program to invoke a cfilabel instruction with the loaded label. However, this approach leads to coarse-grained policies for backward-edge CFI as a return instruction can only hold one label, i.e., all possible different call sites are assigned the same cfilabel instruction. Moreover, they evaluated their scheme only in a simulator (Alpha), while we present real hardware implementations and evaluations on two architecturally different embedded platforms.

Branch regulation as proposed by Kayaalp et al. [12] requires identifying function bounds and a shadow stack to enforce fine-grained CFI. However, their approach suffers from the basic problems of shadow stacks (see Section 3.1). In particular, branch regulation by design does not support stack unwinding and tail jumps.

Another popular hardware-assisted approach to prevent code-reuse attacks is based on monitoring branch history information [13, 17]. These defenses do not really implement CFI in hardware but use the branch information hold in the branch trace store (BTS) and last branch record (LBR) of recent Intel® CPUs. However, they deploy too coarse-grained policies or require an ahead-of-time training phase which is a heuristic approach, typically incomplete, and undesirable in most deployment scenarios.

7. CONCLUSION AND FUTURE WORK

For the first time, we present the implementation and evaluation of a fine-grained hardware-assisted CFI scheme that provides integrity checks for backward edges (returns). Our implementation of HAFIX on Intel® Siskiyou Peak and SPARC LEON3 provides new dedicated CFI instructions that efficiently perform CFI checks in a single cycle. We require minimal changes to the compiler toolchain to emit our new CFI instructions. Our security evaluation demonstrates that HAFIX significantly reduces the code base an adversary can leverage to perform code-reuse attacks. Compared to recently proposed software-based CFI approaches, HAFIX reduces the gadget space to 19.82% with an average performance overhead of only 2%.

In our reference implementation of HAFIX we target bare metal code. Ideally, HAFIX needs to be applied to all code running on the target system, including the operating system. We are currently working on an operating system CFI support module that handles label states for different processes. Specifically, this extension will store and restore labels whenever a context switch occurs.

Another ongoing work concerns the label space for programs that link to several shared libraries. In this case, we need to ensure that our compiler emits unique labels per library. We are also currently exploring new CFI instructions that facilitate hardware-assisted forward-edge CFI.

Although embedded systems rarely deploy just-in-time (JIT) compilers, we plan to explore the feasibility of applying HAFIX to dynamically-generated code. In fact, we believe that our mechanisms to enforce backward-edge CFI can be integrated into JIT compilers since we only require assignment of unique labels with a per-function granularity, and emission of CFI instructions at function prologue, epilogue, and call sites.

Acknowledgments

The authors thank the anonymous reviewers, Per Larsen, and Stephen Crane for their constructive feedback. This work has been co-funded by the German Science Foundation as part of project S2 within the CRC 1119 CROSSING, the European Union's Seventh Framework Programme under grant agreement No. 609611, PRACTICE project, and the Intel CRI for Secure Computing.

8. REFERENCES

[1] Gaisler Research. LEON3 synthesizable processor. http://www.gaisler.com.

[2] Gaisler Research. Bare-C Cross-compiler system (BCC). http://www.gaisler.com/index.php/products/operating-systems/bcc.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1), 2009.

[4] M. Budiu, U. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, 2006.

[5] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security*, CCS '10, 2010.

[6] 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 '15, 2015.

[7] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Annual Design Automation Conference - Special Session: Trusted Mobile Embedded Computing*, DAC '14, 2014.

[8] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX conference on Security*, SSYM'14, 2014.

[9] A. Francillon and C. Castelluccia. Code injection attacks on Harvard-architecture devices. In *ACM Conf. on Computer and Communications Security*, CCS '08, 2008.

[10] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc. *GRLIB IP Core User's Manual*, 2008.

[11] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, S&P '14, 2014.

[12] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Annual International Symposium on Computer Architecture*, ISCA '12, 2012.

[13] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX conference on Security*, SSYM'13, 2013.

[14] J. Rattner. *Extreme scale computing*. ISCA Keynote, 2012.

[15] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conf. on Computer and Communications Security*, CCS '07, 2007.

[16] 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 *USENIX conference on Security*, SSYM'14, 2014.

[17] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, 2012.

[18] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX conference on Security*, SSYM'13, 2013.