Hard edges: Hardware-based Control-Flow Integrity for Embedded Devices

George Christou[1], Giorgos Vasiliadis[1], Elias Athanasopoulos[2], and Sotiris Ioannidis[3]

[1] Foundation for Research and Technology Hellas (FORTH-ICS), Greece
[2] University Of Cyprus, Cyprus
[3] Technical University of Crete (TUC-ECE), Greece
{gchri, gvasil}@ics.forth.gr
athanasopoulos.elias@cs.ucy.ac.cy
sotiris@ece.tuc.gr

Abstract. Control-Flow Integrity (CFI) is a popular technique to defend against State-of-the-Art exploits, by ensuring that every (indirect) control-flow transfer points to a legitimate address and it is part of the Control-flow Graph (CFG) of a program. Enabling CFI in real systems is not straightforward, since in many cases the actual CFG of a program can only be approximated. Even in the case where 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.
In this work, we explore the implementation of a full-featured CFI-enabled Instruction Set Architecture (ISA) on actual hardware. Our new instructions provide the finest possible granularity for both intra-function and inter-function Control-Flow Integrity. We implement hardware-based CFI (HCFI) by modifying a SPARC SoC and evaluate the prototype on an FPGA board by running SPECInt benchmarks instrumented with a fine-grained CFI policy. HCFI can effectively protect applications from code-reuse attacks, while adding less than 1% average runtime and 2% power consumption overhead, making it particularly suitable for embedded systems.

1 Introduction

The diversification of computing systems and the wide adoption of IoT devices that pervade our lives has grown the security and safety concerns in home appliances, enterprise infrastructure and control systems. Typical examples range from traditional IoT environments where data are collected and processed in back-end cloud systems, to more sophisticated, edge-based scenarios where part of processing also occurs in end-devices. Protecting against such cases using software-only solutions is not sufficient, since advanced attacks can modify even the security software itself, thus bypassing any restrictions posed. In addition, the performance overheads of software-based solutions is non-negligible in certain cases. The use of hardware-backed solutions can vitally improve the security of embedded devices, even though this is still challenging due to their limited resources and their intrinsic budget of performance and memory.

At the same time, the exploitation threats are constantly evolving. More than a decade ago, exploiting software was as easy as just simply smashing the stack [16]. An attacker could simply inject code into a vulnerable buffer in the stack and overwrite the return address (of the current stack frame) to point back to their code. Today, this is not possible due to data execution prevention (DEP) mechanisms, however attackers can still exploit software in other ways. For instance, code-reuse attacks, such as Return-Oriented Programming (ROP) [19] and Jump-Oriented Programming (JOP) [6] can potentially take advantage of any vulnerability and transform it to a functional exploit. These techniques do not require any code injections; instead, they re-use existing parts of the program to build the necessary functionality without violating DEP. According to a recent report, more than 80% of the vulnerabilities are exploited using code-reuse attacks [18].

Code randomization techniques [17] are shuffling the location of the code, in order to make code reuse attacks harder to achieve. Still, even a small information leak can reveal all of the process code and bypass any randomization scheme [20]. Instead of hiding the code, another way for stopping exploits is to prevent the execution of any new functionality, by employing Control-Flow Integrity (CFI) techniques [3]. An attacker cannot inject code or introduce any new functionality that is not part of the *legitimate* control-flow graph (CFG). Unfortunately, the majority of existing CFI proposals have still many open issues (related to *accuracy* and *performance*), that hinder its applicability [5].

In this work, we extend our previous hardware-assisted CFI (HCFI) [8] in order to enhance its granularity and flexibility. The implementation of new hardware instructions dedicated for CFI, and the deployment of shadow memory within the processor core, increase the granularity of CFI (especially in forward-edge situations); moreover they cover a couple of intrinsic situations (including the instrumentation of fall-through functions and indirect jumps, such as `switch` statements, within functions). Performance-wise, the implementation in hardware is the optimal choise; our approach adds less than 1% average runtime and 2% power overhead, making it suitable for embedded systems.

Overall, HCFI is a hardware design that offers a CFI solution that is (*i*) *complete*, since it protects both forward and backward edges, (*ii*) *fast*, 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 processor core. Furthermore, we argue that HCFI is the most complete hardware implementation of CFI so far, supporting many problematic cases (such as `setjmp/longjmp`, recursion, fall-through functions and indirect jumps within functions).

2 Background

Control-Flow Integrity (CFI) [3] constraints all indirect branches in a control-flow graph (CFG), which is determined statically before the program execution. In essence, this is achieved by setting a simple set of rules that a program execution flow must adhere to:

1. A call-site "A" can call a function "B" only if the edge (the call itself) is part of the Control-Flow Graph (CFG). This is called Forward-Edge CFI and can easily applied to direct calls, as the only way to modify a direct call is to overwrite the code itself. This is not the case for indirect calls though, where function pointers are typically stored in data regions.
2. A function "B" can only return to the call-site "A" that actually called it, and no other place in the code. This is called Backward-Edge CFI. Backward-edges are, in essence, indirect calls, since they rely on a pointer (return address) to jump to their target.

An attacker cannot inject code or introduce any new functionality that is not part of the *legitimate* control-flow graph (CFG). The majority of existing CFI proposals have still many open issues (related to *accuracy* and *performance overhead*), that hinder its applicability, especially, to embedded devices [5, 22]. For instance, it is not always easy to compute the program's CFG. This is mainly because the source code might not always be available, while even if it does, dynamic code that might be introduced at run-time or the heavy use of function pointers can lead to inconclusive target resolution [5]. This problem has led researchers to develop CFI techniques that are based on a relaxed approximation of the CFG [22], 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 [12]. Approximation of the ideal CFG through code analysis is not always sound, therefore, at least for protecting backward edges, the community has suggested *shadow stacks* [9] - secure memory that stores all return address during function calls. Many research efforts have stressed that shadow stacks are important for securing programs, even when we know the program's CFG with high accuracy [11]. A trivial case is when a function is called by multiple places in the program. According to the CFG, all return locations are legitimate, however only one is actually correct. Moreover, implementing fine grained CFI solely on software, introduces prohibitive performance impact. In the original CFI proposal by Abadi [3], the average performance was 21%. More recent approaches like SafeStack [15], are designed to offer fine grained backward edge protection with minimal overhead. The applications are instrumented during compilation in order to use a different, protected stack for storing control flow variables used in backward edges. However, protecting memory regions using software techniques has been proven ineffective against sophisticated attacks [7, 13].

To overcome there restrictions, hardware-assisted CFI implementations can provide architecturally protected memory regions for storing control-flow variables, while at the same time accelerate significantly any checks required during control-flow transitions; this enables the use of fine-grained CFI even in low-powered devices.

3 Threat Model

Our threat model assumes an attacker that can exploit a vulnerability, either a stack or heap overflow, or use-after-free. 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 [20], 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 blocking the attacker from executing directly data with proper CFI annotation. Both of these are commonplace in today's systems preventing software exploitation.

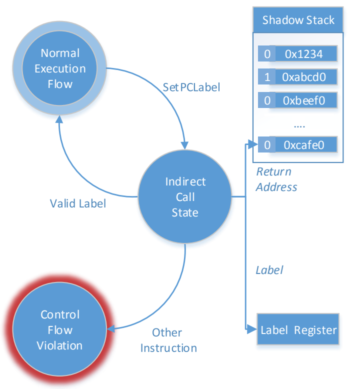
4 Hardware-Enforced Control-Flow Integrity

HCFI enforces the set of CFI rules (described in Section 2) in hardware, while also provide workarounds for certain corner cases. More specifically, a valid call requires that the call site and the destination have been previously acknowledged to be a valid pair in the CFG. A simple way to avoid checking a list of valid pairs for every indirect call, is to group valid pairs with a label. If the label of the source and the destination match, then the edge is legal.

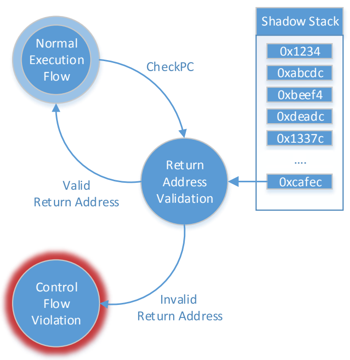
On the contrary, a valid return is typically simpler to validate. Whenever a call takes place, the return address is pushed to the stack. If the address reached after a return, matches the *top* of the stack, the return is valid. This is achieved by also pushing the return address to a new, hidden, stack (namely *shadow stack*), and comparing the return's target to the one stored at the top of the shadow stack. However, this is not the case for the `setjmp`/`longjmp` case, in which a function does not necessarily return to its caller. In particular, `longjmp` never returns to its caller but to its matching `setjmp`.

To support this functionality, the ISA is extended with new instructions (shown in Table 1): two for the instrumentation of the backward edges, two for the forward edges, and two for handling `setjmp`/`longjmp` cases. The instructions are strategically placed, so as to wrap the Control-Flow edges. `SetPC` and `SetPCLabel` are paired with direct and indirect calls respectively, while `CheckPC` is paired with return instructions, and `CheckLabel` is placed in function entry points, if the function is an indirect call target. Finally, `SJCFI` and `LJCFI` are paired with the calls to `setjmp` and `longjmp` themselves. `LJCFI` is placed immediately before the call to longjmp, while `SJCFI` is placed immediately after the call to `setjmp`, so that it will be the first instruction executed after a return from `setjmp`, no matter if `setjmp` or `longjmp` was called.

Finally, given that the design of HCFI does not track stack frames, but specific addresses instead, recursion may result in the same address being pushed



(a) FSM for indirect call instructions



(b) FSM for return instructions

Fig. 1: The basic FSMs for the hardware-based CFI. For return instructions, the target Program Counter is compared with the top value of the stack everytime a `CheckPC` instruction is received and the execution continues normally.

to the shadow stack multiple times. From this observation, a very simple optimization can be implemented; namely, not storing the address when it equals the top of the stack, but instead marking the address at the top as recursive. This effectively negates the spacial requirements of immediate recursion. During `CheckPC` execution, if the top address in the shadow stack is marked as recursive and is the same as the target of the return instruction it will not be popped. If not, the top address will be popped and the target address will be compared with the next top address in the shadow stack. If the two addresses are equal, the execution will continue normally and the top of the shadow stack will be popped (if the address was not marked as recursive). If the addesses are not equal, CheckPC will result in a CFI violation.

Table 1: Instructions needed to support HCFI.

`SetPC`	Pushes the current program counter (PC) in the shadow stack
`CheckPC`	Pops the shadow stack and compares the result with the next PC
`SetPCLabel`	Can push the PC onto the shadow stack and carries a label used to verify forward edges which is stored in a dedicated register (Label Register). Finally, it sets the requirement the next instruction must be a CheckLabel
`CheckLabel`	Carries a label that is compared to the one in the Label Register
`SJCFI`	Sets the environment for a future longjmp and acts as a landing point for an executing one
`LJCFI`	Signifies that a longjmp is underway

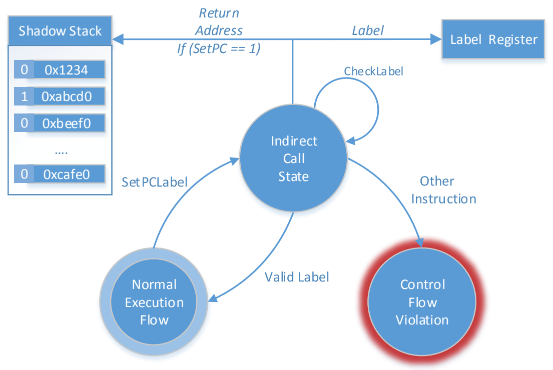


Fig. 2: The extended FSM for Indirect Call States. A `SetPCLabel` instruction is received, the appropriate memory modules are set, and the core enters a state where only `CheckLabel` instructions are accepted. Once a `CheckLabel` instruction with the appropriate label is received, the execution returns to its normal flow.

5 Fine-grained CFI Instrumentation

The instructions presented in Section 4 are created in order to enable a policy agnostic CFI mechanism. Especially for the backward edges, they can easily support the finest possible granularity: by using an architecturally protected shadow stack where only the CFI instructions can modify values, we can ensure that a function will always return to the original call site. However, for forward edges, the granularity is proportional to the effort of analysis performed on the code of the executable. Ideally, every function in the binary will be reachable by a minimum set of indirect call sites. We note that our design can even support more relaxed forward-edge schemes, where indirect call sites can target every function entry point, i.e. by using only one label in the whole binary — this can be practical in cases, where extensive control flow analysis is not feasible.

To allow for finer granularity and flexibility, we make the following modifications to our initial design. Previously, every `CheckLabel` instruction was requiring the Label Register to be set, and hold the correct label. Under the new design, an unset Label Register, or one carrying an incorrect label, does not lead to a violation, as long as the next instruction is also a `CheckLabel`. Also, the `SetPCLabel` instruction can now ommit pushing the PC to the shadow stack, depending on its arguments. Moreover, we allow the instrumentation of indirect branching within the same function. Ignoring `CheckLabel` instructions does not raise security concerns, if the whole binary is instrumented properly. Forward-edge transitions should only be checked during indirect call and branch instructions — during normal execution, the `CheckLabel` instructions do not need to make any checks, since the control-flow is not influenced by data.

5.1 Finer Forward-Edge Granularity

When Control Flow Integrity was first introduced by Abadi et. al. [3], indirect call targets with a common source had to be grouped together. For example, if a call site "A" indirectly called a call target "B", and a call site "C" could indirectly call "D" *and* "B", then both call sites "A" and "C", as well as the call targets "B" and "D", would have to share the same label. This is a usual case in C++ applications where indirect calls, dereference virtual table pointers. Target functions that are common between indirect call sites, will force the use of the same label across a large portion of the application. Thus, the granularity of forward-edge protections become significantly coarser.

In this work, we offer the option to set a unique label for each indirect call site, and add as many `CheckLabels` in the call target as needed. The previous example can now be instrumented with 2 labels in the "B" entry point (one for each indirect call-site). Call site "A" and "C" will carry different labels in their CheckLabel instructions. This has the effect of not allowing call site "A" to jump to "D", which was previously possible. This allows for much finer forward-edge CFI on top of an already powerful design. Figure 2 presents the operation of `CheckLabel` instruction.

Fall-through Functions In many popular libraries, such as GNU libc, there are functions with overlapping code sections [4]. In such cases, the execution of a function falls-through into another function's entry point (without using branch instructions). If these functions are possible targets of indirect call instructions, they should be instrumented with `CheckLabel` instructions, otherwise even if the indirect transition is valid it will result to a CFI violation. Since `CheckLabels` do not cause a CFI violation when the processor is not in indirect jump state, they are just ignored during execution. Thus, when a function falls through, the execution of the inner function's `CheckLabel` instructions will not result in a CFI violation. This allows for fall-through functions to be instrumented like regular functions.

Intra-Function Forward-Edges Most CFI schemes do not take into account indirect branches, targeting addresses within the same function. For example, large `switch` statements are usually compiled to jump tables in order to reduce the code size of the binary. In these cases the address of each case is stored in a jump table. At runtime, the result of the `switch` statement is used in an indirect jump in order to dereference the jump table at the appropriate index. Thus, instead of emitting absolute jumps for every possible statement result, the compiler emits a single indirect jump that uses the statement result as an index in the jump table. In our design we offer the capability to instrument those indirect jumps in order to ensure that the target address is the entry point of one of the cases. Each indirect jump will be instrumented with a `SetPCLabel` instruction that will not push a return address in the shadow stack (i.e. `SetPC` bit is '0'), and the entry points of each *case* basic block will be instrumented with the appropriate `CheckLabel` instruction. Every `switch` statement in the binary should use a different label for better granularity.

6 Prototype Implementation

To implement the hardware-based CFI described in the previous sections, we extended the Leon3 SPARC V8 processor, which is a 32-bit open-source synthesizable processor. Overall, the additions to the core can be grouped in the following two categories: (*i*) Memory Components and (*ii*) CFI Pipeline.

6.1 Memory Components

The following new memory components are deployed in the Register File of the core:

– A 256*32 bit dual-port Block RAM was used for the Shadow Stack.
– A 256*8 bit single-port Block RAM was used for the `setjmp` and `longjmp` support (SJLJRAM).
– A 18 bit register was used to store the label for forward edge validation (Label Register).
– A 256*1 bit array helped us optimize recursive calls (Recursion Array).

6.2 CFI Pipeline

Our instructions enter the Integer Unit's (IU) pipeline as usual, however they do not interfere with it. We have developed a new pipeline within the IU (CFI Pipeline) that operates in parallel and provides the functionality required everytime the instructions are decoded.

– `SetPC` first tops the Shadow Stack and compares it to the current Program Counter (PC). If the memory addresses match, the Recursion Array is set; otherwise, the address is pushed onto the shadow stack. In case the Shadow Stack is full a *Full* violation is raised.
– `SetPCLabel` is in essence two instructions, meaning that it acts exactly as a `SetPC` and what could be described as a `SetLabel`. The `SetPC` functionality works only if the 25th LS bit of the instruction is set. Regardless of the `SetPC` functionality, the Label carried in its 18 LS bits is written to the Label Register, and the CFI Pipeline transitions to the `SetLabel` state. This mandates that only `CheckLabel` instructions can be executed, until one with the correct label is issued. If any other instruction is issued, a *Control Flow* violation is raised.
– `CheckLabel` compares the Label carried in its 18 LS bits to the label stored in the Label Register, if the CFI Pipeline is in the `SetLabel` state. Otherwise, it is ignored and acts as a `nop`. If the comparison holds, the Label Register is reset and the pipeline transists from SetLabel state to normal execution. If not, the execution continues, but if an instruction other than checklabel is issued, a *Control Flow* violation will be raised.

- `CheckPC` first checks the Shadow Stack; if it is empty, an *Empty* violation is raised. Otherwise, it tops the Shadow Stack, increments the value by four (one instruction) and compares it to the next PC. If the addresses match and the equivalent recursion bit is not set, the Shadow Stack is popped. If the addresses did not match but the recursion bit is set, the address is popped and another comparison is performed with the next value. Again, if they match and the top value is not recursive, it is popped. If the first comparison failed and the top address was not recursive, or if both comparisons failed, a *PC Mismatch* violation is raised.
- `SJCFI` changes its functionality depending on whether the CFI Pipeline is in the `longjmp` state. If it is not, it writes the current depth of the Shadow Stack to the SJLJRAM. The address is provided by a label it carries on its 8 LS bits. Otherwise, it uses the same label to read the address from the SJLJRAM and set the Shadow Stack to that depth. The Shadow Stack will not allow an index higher than the current, so that previously popped addresses cannot be abused. The CFI Pipeline returns to its default state.
- `LJCFI` sets the pipeline in the `longjmp` state until an `SJCFI` instruction is executed.

7 Performance Evaluation

We synthesize and program our new design, based on the 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. Since we are targeting embedded systems, we run all tests without an operating system present. We instrumented most of the SpecInt2000 suite and a few microprocessor benchmarks, namely Coremark, Dhrystone, and Matmul.

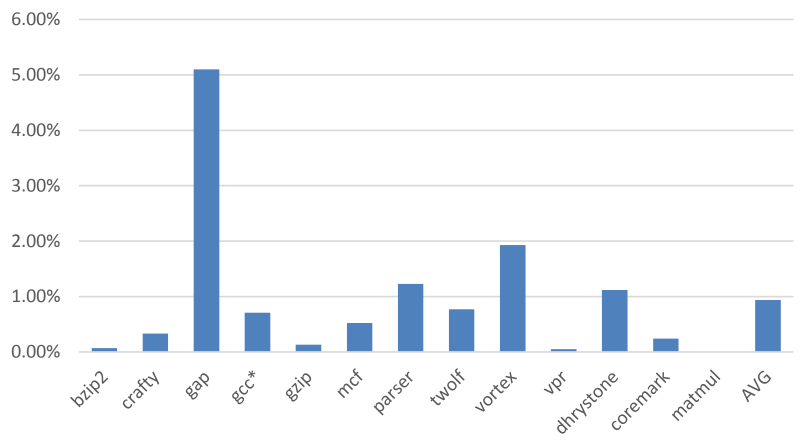


Fig. 3: The runtime overhead measured with our implementation.

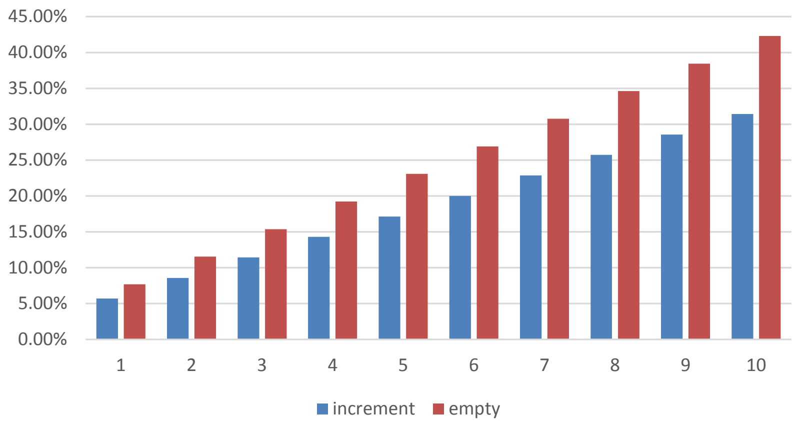


Fig. 4: The runtime overhead added by using 1-10 labels on an empty function or a function that increments a value.

Runtime Overhead When instrumenting only calls (both direct and indirect) and returns, the average overhead lies at a little under 1% as shown in Figure 3. In the case of `gap` benchmark, the reported overhead is the result of a tight loop executing a multitude of indirect calls to relatively small functions.

We also run two series of microbenchmarks to see the effect of adding multiple labels to a function. We did this by executing a tight loop with an indirect call to one of two functions. The first was an empty function, which results in three assembly instructions. The second was a function that incremented a global variable, this has a body of ten instructions. We added one to ten labels on the function entry points. With these benchmarks we can find the maximum percentage of runtime overhead imposed when a function is called indirectly with CFI instrumentation. We present our results in Figure 4. In our previous design the maximum runtime overhead that could be imposed is the same as the overhead reported for the empty function with only one label. The runtime overhead is relative to the number of indirect call sites that can point to each function (i.e. the number of labels in the entry point) and the number of instructions in the function. In large functions, CFI instructions will account for a small percentage of the function's code. Thus, we expect that the performance overhead will be significantly less in real world applications. By also instrumenting indirect jumps, the overhead can increase; even though this depends on the total number of indirect branches that the program uses. For example, forward-edge protection in the jump table implementation of switch statements, can be accomplished with the execution of just two additional instructions. In our new design, the granularity of forward-edge can be adjusted, i.e. use the same labels in some indirect call sites in order to reduce the number of labels in function entry points. Thus, application designers can opt to reduce forward-edge granularity in order to favor performance.

Hardware Overhead We implemented our design initially without `longjmp` support and the recursion optimization. The resulting area overhead, as detailed by the reports of the Xilinx tools used to synthesize the design, is very low, using an additional 0.65% registers and 0.81% LUTs (look-up tables). The area overhead increases significantly to 2.52% registers and 2.55% LUTs, when placing the `longjmp` support and the recursion optimization.

Power Consumption We measure the power consumption of our design using the Xilinx XPower Analyzer tool. For the unmodified design the tool reported 6072.11 mW power consumption. The required modifications for the CFI instructions increase the power consumption to 6149.32 mW. The full fledged design with CFI and SJ/LJ support has a power consumption of 6176.92 mW. The results indicate that the power consumption overhead is about 1.2%, which increases to 1.7% when adding `longjump` support.

8 Related Work

CFI is the base of many proposed mitigation techniques in the literature. 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 strategies proposed in the literature and the industry as well as their limitations. Davi et al. [10] proposed HAFIX, a system for protecting backward edges based on active set CFI. HAFIX deploys dedicated, hidden memory elements for storing critical information. Their implementation utilizes labels to mark functions as active call sites. Labels are used as index in a bitmap, which dictates if a function is active or inactive. A return can only point to an `active` function. However, it has been proven that this notion is very relaxed and can be circumvented [21]. In our design we use an architecturally protected shadow stack, a technique considered to be the state of the art for protecting beackward edges. Moreover, our design offers forward edge protection. HAFIX proposes the use of software techniques for protecting forward-edges.

Intel plans to include Control-flow Enforcement Technology (CET) [1] in future processors. In CET a shadow stack is defined in order to protect backward-edge control flow transfers in a manner similar to our design. With regards to forward-edge control flow transfers `ENDBRANCH` instruction is used to mark the legitimate landing points for call and indirect jump instructions within the applications code. However, an indirect jump can point to any `ENDBRANCH`. In comparison, HCFI can support multiple labels in every function entry offering per indirect call-site granularity for forward edges. ARM presented Pointer Authentication Code (PAC) [2]. This mechanism utilizes cryptographic primitives (hashing) in order to verify that the control flow pointers are not corrupted before using them. The pointer authentication code (PAC) of each control flow pointer is stored in the unused bits of the pointer (i.e. 24 MS bits of the pointer). Each process has a unique key which is used in order to calculate and authenticate the control flow pointers. The encryption algorithm used is QARMA. This technology has been already deployed in Apple products with ARMv8.3 cores. A recent study from Googles project zero identified several vulnerabilities in this technology [14]. Pointer authentication can offer similar levels of protection with our design. However, the use of cryptographic primitives in PAC instructions imposes signifficantly more overhead in terms of performance and area compared to our design.

9 Conclusions

In this paper, we designed, implemented and evaluated a flexible and policy-agnostic Control-Flow Integrity Instruction Set Extension. Our extensions introduced less than 1% runtime overhead on average and less than 2% increase in power consumption, will only imposing very little overhead in terms of additional hardware circuitry (less than 2.55%). Our plan for the future is to extend our implementation to support multi-threading. While our forward-edge protections can be easily deployed in multi-threaded applications, for protecting backward-edges a single shadow stack is not enough. We plan to implement a new technique that allocates memory pages for each thread's shadow stack.

Acknowledgments
This work was supported by the projects CONCORDIA, C4IIoT, Cyrene and IntellIoT, funded by the European Commission under Grant Agreements No. 830927, No. 833828, No 952690 and No. 957218. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein. The authors would like to thank Nikolaos Christoulakis for his contribution during the implementation of HCFI.

References

1. Control-flow Enforcement Technology Preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf` (2016)
2. Pointer Authentication on ARMv8.3. `https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf` (2017)
3. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM CCS (2005)
4. Agadakos, I., Jin, D., Williams-King, D., Kemerlis, V.P., Portokalidis, G.: Nibbler: Debloating binary shared libraries. In: Proceedings of the 35th Annual Computer Security Applications Conference. pp. 70–83 (2019)
5. Athanasakis, M., Athanasopoulos, E., Polychronakis, M., Portokalidis, G., Ioannidis, S.: The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In: NDSS. The Internet Society (2015)
6. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (2011)

7. Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R.: Control-flow bending: On the effectiveness of control-flow integrity. In: USENIX Security (2015)
8. Christoulakis, N., Christou, G., Athanasopoulos, E., Ioannidis, S.: HCFI: Hardware-Enforced Control-Flow Integrity. In: Proceedings of the 6th ACM Conference on Data and Application Security and Privacy. CODASPY '16 (2016)
9. Dang, T.H., Maniatis, P., Wagner, D.: The performance cost of shadow stacks and stack canaries. In: ACM Symposium on Information, Computer and Communications Security, ASIACCS. vol. 15 (2015)
10. Davi, L., Hanreich, M., Paul, D., Sadeghi, A.R., Koeberl, P., Sullivan, D., Arias, O., Jin, Y.: Hafix: hardware-assisted flow integrity extension. In: Proceedings of the 52nd Annual Design Automation Conference. p. 74. ACM (2015)
11. Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., Sidiroglou-Douskos, S.: Control jujutsu: On the weaknesses of fine-grained control flow integrity. CCS (2015)
12. Göktaş, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: Overcoming control-flow integrity. In: IEEE Symposium on Security and Privacy (2014)
13. Göktaş, E., Economopoulos, A., Gawlik, R., Athanasopoulos, E., Portokalidis, G., Bos, H.: Bypassing Clang's SafeStack for fun and profit. Black Hat Europe (2016)
14. Google Project Zero: Examining Pointer Authentication on the iPhone XS. `https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html`
15. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-Pointer Integrity. In: USENIX OSDI (2014)
16. One, A.: Smashing the stack for fun and profit. Phrack magazine **7**(49), 365 (1996)
17. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In: Proceedings of the IEEE Symposium on Security and Privacy (2012)
18. Rains, T., Miller, M., Weston, D.: Exploitation trends: From potential risk to actual risk. In: RSA Conference (2015)
19. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC) **15**(1), 2 (2012)
20. Snow, K.Z., Davi, L., Dmitrienko, A., Liebchen, C., Monrose, F., Sadeghi, A.R.: 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)
21. Theodorides, M., Wagner, D.: Breaking active-set backward-edge CFI. In: IEEE International Symposium on Hardware Oriented Security and Trust (2017)
22. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: USENIX Security (2013)