Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation

Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi
Intel Collaborative Research Institute for Secure Computing (ICRI-SC) at
Technische Universität Darmstadt, Germany
{lucas.davi,ahmad.sadeghi}@trust.cased.de
patrick.koeberl@intel.com

ABSTRACT

Embedded systems have become pervasive and are built into a vast number of devices such as sensors, vehicles, mobile and wearable devices. However, due to resource constraints, they fail to provide sufficient security, and are particularly vulnerable to runtime attacks (code injection and ROP). Previous works have proposed the enforcement of control-flow integrity (CFI) as a general defense against runtime attacks. However, existing solutions either suffer from performance overhead or only enforce coarse-grain CFI policies that a sophisticated adversary can undermine. In this paper, we tackle these limitations and present the design of novel security hardware mechanisms to enable fine-grained CFI checks. Our CFI proposal is based on a state model and a per-function CFI label approach. In particular, our CFI policies ensure that function returns can only transfer control to active call sides (i.e., return landing pads of functions currently executing). Further, we restrict indirect calls to target the beginning of a function, and lastly, deploy behavioral heuristics for indirect jumps.

1. INTRODUCTION

Embedded systems have radically evolved over the last years and are permeating our information societies. They can be found almost everywhere, in sensors, RFID tags, smart cards, mobile and wearable devices, vehicles, industry control systems, etc. With the increasing deployment and integration of embedded systems in safety, security and privacy sensitive applications as well as critical infrastructures, we are facing new challenges with respect to their trustworthiness, beyond that of traditional platforms [11].

Resource constraints are typically a challenge in embedded systems (computational, storage, energy, etc.) [15]. As a consequence, security is usually only introduced if the corresponding resource usage is minimal. A second challenge is that embedded systems are usually programmed using native (unsafe) programming languages such as C or assembly language. As a consequence, they frequently suffer from vulnerabilities that can be exploited by runtime attacks. In general, such runtime attacks divert the desired control-flow of (embedded system) software by exploiting bugs such as buffer overflows. After gaining control over the program flow the adversary can inject malicious code to be executed (*code injection* [3]), or re-use and combine existing code pieces (gadgets) that are already residing in program memory (e.g., in linked libraries) to implement the desired malicious functionality (*return-oriented programming* [16]).

One promising and general (software-based) defense technique against runtime attacks is the enforcement of *control-flow integrity* (CFI) [1]. In principle, CFI ensures that an application only executes according to a pre-determined control-flow graph (CFG). Since code injection and return-oriented programming (ROP) attacks result in a CFG deviation, CFI detects the malicious flow and prevents the attack. CFI can be realized as a compiler extension [9] or as a binary rewriting module [1].

Although CFI is an effective and fundamental defense against runtime attacks, current solutions suffer from several limitations such as performance overhead and requiring presence of debug symbols. Since CFI implementations on desktop systems suffer from performance issues, the situation will be exacerbated when applied to embedded system software. To tackle some of these shortcomings new CFI solutions have been presented recently [12, 19] that aim at coarse-grained CFI to improve efficiency but under weaker adversary models (see Section 3).

Goals and Contributions.

Our goal in this paper is to tackle the challenge of designing a *fine-grained, hardware-based CFI defense* for embedded systems that in particular mitigates the crucial class of return-oriented programing (ROP) attacks. In summary, our CFI design is based on a state model and per-function label approach to enforce the following CFI policies:

- **Function Calls**: After a function call we enforce the processor to switch to a new state called *function entry*. In this state, the processor only accepts our new CFI instruction (`CFIBR label`). These instructions are inserted by the compiler at the beginning of each function. Hence, calls can only target a valid function entry. Moreover, the label is stored in a CFI Label State memory to keep track of currently executing functions.

DAC '14, Jun 01-05 2014, San Francisco, CA, USA

- **Function Returns**: Our CFI policy for returns enforces that a return only targets a valid return landing pad of a function whose label is currently active. This is achieved by using a new CFI instruction (`CFIRET label`) and indexing our CFI Label State memory.

- **Indirect Jumps**: For these instructions, we use behavioral heuristics in a window of five consecutive indirect jumps. Our heuristics cover typical patterns of ROP attacks (e.g., number of direct branches issued, as well as stack push and pop instructions).

2. RUNTIME ATTACK THREAT MODEL

Runtime attacks exploit vulnerabilities in the software running on the targeted embedded system. Relevant vulnerabilities are memory errors such as stack, heap, or integer overflows [13]. Such vulnerabilities are likely to be discovered since most embedded system software is implemented in unsafe languages like C or assembly, and large numbers of memory errors are reported on an ongoing basis (see e.g., NIST vulnerability database CWE category: buffer errors[1]).

The adversary exploits vulnerable code by providing a malicious input that contains (i) data to overflow the buffer, (ii) the payload that performs the desired malicious operations, and (iii) new control-flow information (e.g., a new return address) used to hijack the execution flow and transfer control to the payload. The payload either consists of a malicious code (i.e., assembler instructions) or a number of pointers that point to code pieces in linked libraries[2]. The former case represents a code injection attack which is typically defeated by enforcing the non-executable memory principle (DEP or W⊕X) used on many platforms today. In contrast, the latter case refers to return-oriented programming (ROP) attacks re-using short code sequences (gadgets) that are chained together and executed. These gadgets can be re-used from any software that is accessible in virtual memory. In contrast, the latter case refers to return-oriented programming (ROP) attacks that re-use short code sequences (gadgets) that are chained together and executed. These gadgets can be re-used from any software that is accessible in memory.

Hence, we *assume* that the adversary does not aim at injecting malicious code or new malicious applications (which is an orthogonal topic), but it is rather able to launch ROP attacks. Defending against these attacks by introducing a new hardware architecture is the main goal of this paper. Since ROP attacks are prevalent in today's exploits, we will take a deeper look how these attacks work internally.

Figure 1 depicts a typical memory layout of a ROP attack. The memory area controlled by the adversary contains the ROP payload, where the payload consists of several return addresses each pointing to a ROP sequence residing in the address space of the application. All these sequences either terminate in a return [16] or indirect jump/call [5] instruction. The payload can also contain data words that the invoked ROP sequences load into registers via stack POP instructions (see ROP Sequence 2). The stack pointer register (on x86: `ESP`) plays a crucial role in each ROP attack, because it dictates which ROP sequence needs to be executed next. At the beginning of the attack, it points to the first

[1] `http://web.nvd.nist.gov/view/vuln/search-advanced`
[2] A combination of both payload types is also possible.

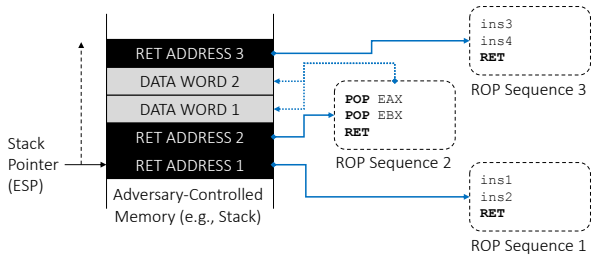

Figure 1: Memory Snapshot of a ROP Attack

return address of the payload. When ROP Sequence 1 executes the final return instruction (RET), the stack pointer is automatically advanced by one memory word and control is transferred to the next ROP sequence (ROP Sequence 2).

We also *assume* that the adversary is able to bypass ASLR (Address Space Layout Randomization), a defense technique deployed in today's systems that randomizes the base address of libraries and executables. As a result ASLR randomizes the start addresses of those code sequences the adversary attempts to invoke in a ROP attack. However, ASLR is vulnerable to memory disclosure attacks, which reveal runtime addresses to the adversary. Such attacks are capable of circumventing even fine-grained ASLR schemes [17].

3. RELATED WORK

Over the last two decades a number of defense techniques have been proposed to mitigate runtime attacks. In contrast to many ad-hoc solutions the most general defense, and the focus of this paper, is based on the principle of control-flow integrity (CFI) [1], that tackles runtime attacks regardless of their specific origin.

3.1 Software-Based CFI Approaches

The original CFI proposal [1] deploys static binary instrumentation to derive the control-flow graph (CFG) of an application binary and extends it with CFI checks. It aims at fine-grained protection by ensuring that indirect jumps and calls only target an address that maps to a valid CFG path, and monitoring that returns only use return addresses that are held on a so-called shadow stack. For the latter, function calls are instrumented to always copy the return address on the shadow stack. One of the main practical limitations of CFI [1] is that it induces a performance overhead on average by 21% and requires debug symbols. Follow-up works tackled these limitations in terms of performance and the necessary extra information [19], and also showed that CFI enforcement can be realized for ARM-based smartphones [6, 9]. However, the solution in [6] still suffers from performance penalties. Furthermore, the CFI policy used for return instructions in [19] only provides a coarse-grained check, because it allows returns to target any instruction that follows a call instruction.

3.2 Hardware-Assisted CFI Approaches

Several approaches leveraged (or introduced new) hardware-based mechanisms to mitigate runtime attacks. kBouncer uses the Last Branch Recording (LBR) history table of recent Intel processors [12]. It adds hooks into API call sites, and once these are triggered at runtime, it validates the LBR

entries based on a CFI policy. However, this policy is coarse-grained: returns are allowed to target any instruction after a call, and indirect jumps and calls are not monitored. Instead, kBouncer counts the number of instructions executed between two indirect branches to identify ROP sequences.

CFIMon uses performance counters and Intel's Branch Trace Store (BTS) to detect control-flow deviations [18]. Zhang et al. detect program execution anomalies based on a new hardware architecture that validates all branch instructions [20]. Dynamic Sequence Checking validates the hamming distance between basic blocks against known values using a dedicated runtime execution monitor hardware module [10]. However, some of these approaches require an offline training phase [18, 20] and all assume a precise and static control-flow graph (CFG). Deriving a complete CFG is hardly possible given the complexity and dynamic nature of modern programs. In particular, statically determining valid return addresses leads to coarse-grained CFI policies as described in [1].

The CFI design presented in this paper is most closely related to the hardware-assisted CFI design presented by Budiu et al. [4]. In their work, new hardware CFI instructions are introduced to enforce label checks on each indirect branch. For this, each branch target is annotated with a label instruction (`cfilabel L`), and every indirect branch is replaced by a corresponding CFI instruction, e.g., `jmpc reg,L`. The latter CFI instruction jumps to the address specified in `reg` and at the same time sets a label `L` in a dedicated (new) CFI register. After the indirect branch has been executed, the processor changes state such that `cfilabel` is the only permissible next instruction. In particular, the state will only change back to the ordinary execution state after a `cfilabel` instruction has been executed using exactly the label `L` that has been stored in the CFI register by the indirect branch. The main drawback of this approach is that it leads to coarse-grained CFI policies for return instructions. Note that a subroutine can be called by different callers, but the return of the subroutine can only use one label, e.g., `retc L1`. Hence, in order to preserve the program semantics at each possible call side, the compiler needs to insert a label instruction using `L1`. This allows the adversary to choose to which call side he wants to return. Moreover, another related problem arises for those indirect calls that can potentially target many diverse functions. As an example, consider an indirect call that can possibly target 200 functions (which is not an artificial scenario even if source code is available, see e.g., [2]). To ensure fine-grained CFI, one would need to assign a unique label for each of the 200 function return instructions. As such, the compiler needs to add 200 corresponding label instructions at the call side (the instruction after the indirect call), i.e., 200 `cfilabel L1-L200` instructions. This leads to a significant space and performance overhead. Given these problems, using the approach presented in [4] will with high probability lead to a coarse-grained CFI policy where identical labels will be used for a subset of returns.

To summarize, many software and hardware-based CFI approaches rely on coarse-grained return policies allowing a return to target any call-preceded instruction. As we will further analyze in Section 4 this allows an adversary to bypass these defenses and mount ROP attacks. As a consequence, we aim towards a more effective and at the same time efficient protection against ROP attacks.

4. PROBLEMS OF COARSE-GRAINED CFI

As mentioned in Section 3, many CFI schemes allow a return instruction to target any valid call side, i.e., any instruction that follows after a call instruction. While it is widely believed that this protection level is already sufficient to thwart ROP attacks (see e.g., [4, 19, 12]), we argue that this only negligibly raises the bar for an adversary. For this, we performed a small experiment, where we statically analyzed the standard UNIX C library `libc.so` and searched for valid call-preceded gadgets. Our analysis reveals that there are 2,242 valid call-preceded sequences each terminating in an indirect branch in the 1.6MB large `libc.so` library. In particular, we only consider sequences that contain at most 10 instructions[3]. In other words, the adversary can target more than 2,000 locations when he exploits a single return instruction. Based on this large set of ROP sequences we constructed a proof-of-concept ROP exploit that launches a new shell to the adversary. Our proof-of-concept attack demonstrates the ineffectiveness of coarse-grained CFI, and confirms the attacks recently presented against Windows EMET and CFI for COTS binaries [7, 8].

5. DESIGN OF HARDWARE-ASSISTED CFI

In this section, we introduce the design and implementation of our novel security hardware architecture that provides fine-grained protection against ROP attacks based on control-flow integrity (CFI).

Our basic design modelled as a state machine is shown in Figure 2. The main idea of our protection mechanism is to enforce CFI based on label *state*, and decouple source from destination labels (e.g., as used in [4]). For this we distinguish between four states: (0) for ordinary program execution, (1) function entry, (2) function exit, and (3) CFI exception state when an attack is detected. To enforce our CFI label approach we envision the introduction of two new CFI label instructions, namely `CFIBR` and `CFIRET`. As the names imply, we use one label instruction for function calls and another one for returns. This distinction allows us to deploy different policies for calls and returns, and at the same time ensures that an indirect call cannot target a label instruction used for returns and vice versa.

Call Instrumentation.

In state 0, three types of indirect branches are of interest for CFI. First, direct and indirect calls will lead to a transition from state 0 to state 1 (function entry). In this state, we only allow the program to use a `CFIBR` instruction. Any other instruction will lead to a CFI violation and transition to state 3. Each `CFIBR` instruction contains a *label*, which is hard-coded as an immediate. Specifically, we use labels on a per-function level. In other words, every function in the program is assigned one unique *label* and we insert a `CFIBR` instruction at the beginning of each function.

The effect of a `CFIBR` is twofold: first, it loads the used label in a dedicated and isolated memory storage that we refer to as *CFI Label State*. This operation effectively activates a label, i.e., it indicates that a function has been entered. The second effect is that after `CFIBR` has been executed, the processor changes back to state 0. Our mechanism ensures that an indirect call must target a `CFIBR` instruction. Since

[3]Larger sequences are typically not useful for ROP attacks since they induce too many side-effects.

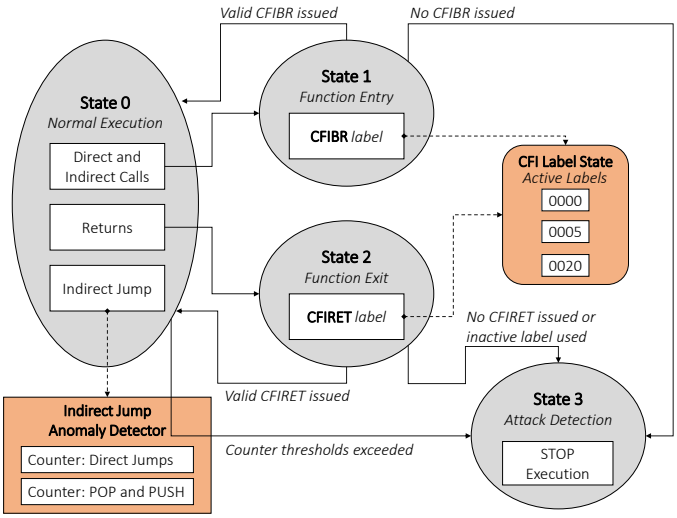

Figure 2: Design of Hardware-Enforced ROP Detection

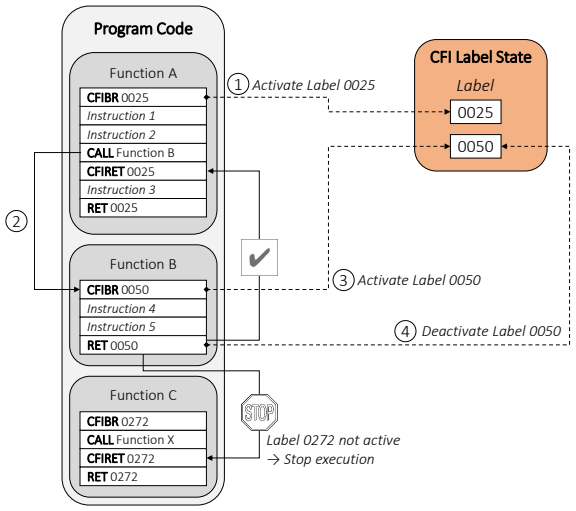

Figure 3: CFI policy for function returns

these are placed at each function entry, we prevent the adversary from jumping into the middle of a function. This basic check is along the lines of existing CFI approaches. Our main security improvement compared to previous works is our instrumentation for function returns.

Return Instrumentation.

Whenever a return instruction is issued by the program, we exchange the state from 0 to state 2. Similar to state 1, we only allow the execution of `CFIRET` instruction in this state. Furthermore, state 2 enforces a CFI policy ensuring that only those `CFIRET` instructions with an active *label* in our CFI Label State memory are executed. This policy ensures that function returns can only target a function that is currently executing. To support this scheme, the CFI compiler emits `CFIRET` instructions at all valid call sides (i.e., at all instructions following a call instruction) using exactly the same *label* that has been assigned to the function.

A detailed flow of the return protection is shown in Figure 3. It shows a sample program consisting of three functions: A, B, and C. Each function is assigned a label, e.g., label `0025` is used for Function A. As described above this label is written to our CFI Label State memory through the `CFIBR` instruction at the beginning of each function (step 1).

Note that Function A performs a subroutine call to Function B (step 2). Hence, the processor switches to state 1, and the `CFIBR` of Function B activates label `0050` in the CFI Label State memory (step 3). The critical point with regards to CFI is the function return in Function B. Potentially, the adversary could have manipulated the return address to hijack the execution flow. To prevent this attack, we apply our CFI policy for returns ensuring that that a return can only target a `CFIRET` instruction using an *active* label. In the example shown in Figure 3, our CFI policy is preserved when the program returns to Function A, because at its call side the CFI compiler has emitted a `CFIRET 0025` instruction. Consequently, the adversary has no chance to redirect the control-flow to the call side of Function C, since label `0272` is not active. Recall that coarse-grained CFI protection schemes that only deploy one label for returns would have not prohibited this malicious execution flow.

Since we activate labels, we also need to deactivate them at the time the function returns. This can be achieved in two ways: one option is to embed the label in the return instruction. Alternatively, we could introduce a separate new instruction that performs the deactivation and place it before the return instruction, e.g., `DEACT label`.

Indirect Jump Instrumentation.

Although most existing exploits today deploy ROP attacks that misuse return instructions to chain gadgets together, there is also the possibility of mounting ROP attacks using indirect jumps [5]. Although these attacks are more challenging to execute, since there are not as many sequences available as for returns, they are harder to prevent due to the fact that many indirect jumps cannot be resolved prior to execution, i.e., their set of target addresses are hard to predict. Hence, we leverage in our design a heuristic behavioral-based approach that does not rely on perfect resolving of indirect branch targets. Specifically, we

keep track of several counters in the window of five indirect jumps. One counter keeps track of the number of direct branches executed between our sliding window. This is a reliable ROP attack indicator, because ROP attacks today rarely employ direct branches but rely on indirect branches whose target addresses can be controlled. In contrast, direct branches are hard-coded and in most cases will lead to a loss of control in the ROP chain. Another two counters are deployed to keep track of stack pushes and pops. Typically, ROP attacks do not use push instructions [16] as these would typically destroy the ROP chain by overwriting pointers. That said, if many pop but only a few push instructions are issued in our window, we have another indicator that a ROP attack is currently executing.

Effectiveness.

Our design on hardware-assisted CFI effectively mitigates ROP attacks. In contrast to previous work on coarse-grained CFI policies for returns (e.g., [12, 19]), we enforce that returns can only target a call site of a function whose label is currently active. This effectively prevents ROP attacks that invoke call-preceded gadgets [8] of which more than 2,000 sequences exist only in the standard UNIX C library libc (see Section 4). Moreover, our approach to return instrumentation is more efficient and easier to maintain than a shadow stack, as in [1], because we do not need to perform a cumbersome and slow return address match validation.

To prevent ROP attacks from jumping into the middle of a function, we enforce that call instructions target a valid function start. This would still allow an adversary to launch pure return-into-libc attacks where only entire functions are invoked by exploiting indirect calls. Note that such attacks are not specific to our design but apply to previous CFI solutions as well. However, return-into-libc attacks typically still need to invoke at least some ROP sequences to prepare function arguments, which is detected by our approach. Moreover, assuming a precise control-flow graph as in [1, 4], our design can be easily leveraged to enforce fine-grained CFI checks for indirect calls. We would only need to load all valid labels of an indirect call in a second label memory area (*Call Labels*) and, restrict the subsequent `CFIBR L` instruction to use a label that can be found in *Call Labels*.

For our indirect jump heuristics, we already tested several thresholds using SPEC CPU2006 benchmarks. Our experiments reveal that between a sliding window of five indirect jumps there are never execution traces that contain less than 3 direct jumps and 3 push instructions. Although heuristic-based approaches will never provide an ultimate solution to runtime attack prevention, they can significantly raise the bar for control-flow attacks. This is the case in our design, where we provide fine-grained protection of returns combined with heuristic-based policies for indirect jumps. Another alternative we are currently investigating is a new CFI policy for indirect jumps that ensures indirect jumps can only target a function whose label is active. Since indirect jumps typically remain in the boundaries of the function where they are executing[4], it seems to be a promising direction that we will explore. In particular, we will investigate whether such a strict CFI policy will raise false positives. For the time being, we employ the heuristics described above.

[4]There are some known exceptions such as stub code for calls to external library functions in the Linux PLT (Procedure Linkage Table) section. To support these indirect jumps, we could enforce that indirect jumps are allowed to target functions whose label is active or whose next instruction is a `CFIBR L`.

Instruction	Semantics
`CFIBASE baseaddr`	cfi_base_reg := baseaddr
`CFIBR label`	[cfi_base_reg + label] := 1
`CFIRET label`	if [cfi_base_reg + label] != 1 then STOP
`RET label`	[cfi_base_reg + label] := 0
`DEACT label`	[cfi_base_reg + label] := 0

Table 1: Instruction extensions and semantics

6. IMPLEMENTATION PLAN

We intend to implement our hardware-assisted CFI on the Intel Siskiyou Peak research architecture [14]. Siskiyou Peak is a 32-bit, 5 stage pipeline, single-issue processor design targeted primarily at embedded applications. The processor is organized as a Harvard architecture with separate buses for instruction, data and memory-mapped IO spaces and is fully synthesizable.

CFI Label State Table.

We propose that the CFI label state table be embedded in the data segment of the executing program with an associated base register (`cfi_base_reg`) defining the appropriate data segment offset. This approach supports multi-tasking environments where the appropriate offset can be managed by an OS on a task-by-task basis. The CFI label embedded in `CFIBR`, `CFIRET` and `RET` or `DEACT` would provide the index into the table. Embedding CFI label state into the program will result in increased memory bandwidth and increased cache pressure in multi-level memory architectures. While this is a disadvantage in those contexts, for deeply embedded processors at the lower end of the performance scale, where simple physical addressing and Tightly Coupled Memories (TCMs) are typical, this would be less of an issue. Table management could approach single cycle performance in these usages. Other trade-offs are possible in relation to table data representation, for example, packing 32 label states into a 32-bit memory word at the expense of performance and additional logic complexity.

ISA Extensions.

The Siskiyou Peak ISA is a subset of the Intel Architeture (IA) instruction set. As detailed in previous sections the ISA will be extended with `CFIBR`, `CFIRET`, and either a modified `RET` or new `DEACT` instruction. Table 1 shows the new instruction semantics. The base address of the CFI state table, `cfi_base_reg`, is set using the `CFIBASE` instruction.

Exception Handling.

A necessary design decision regarding CFI in general pertains to the handling of detected CFI violations. In our scheme we intend to raise a processor exception when transitioning to state 3 of Figure 2. The address of the instruction which raised the exception will be recorded and execution will transfer to an exception handler routine which can decide on the correct course of action. This approach affords flexibility in how CFI violations are dealt with and provides

a means to restart program execution with no loss of continuity. This may be appropriate for violations resulting from the indirect jump heuristics.

Indirect Jump Detection.

It is envisaged the the heuristics associated with indirect jump detection will be implemented in an fully autonomous manner with software only responsible for configuring the appropriate indirect jump window size and thresholds for direct jumps and push and pop stack operations. More specifically, indirect jumps will be counted and when the window size, e.g. 5, is reached the direct jump and push and pop counters will be compared against configurable thresholds. If these thresholds are exceeded, a transition to state 3 in Figure 2 will occur indicating a ROP attack based on indirect jumps. Conversely, if no thresholds have been exceeded the counters will be reset and monitoring will continue.

7. SUMMARY AND CONCLUSION

Runtime attacks such as return-oriented programming constitute a powerful attack vector on a broad range of processor architectures, including embedded systems. With no doubt, the principle of control-flow integrity (CFI) is a robust and general defense technique to effectively mitigate these attacks. In particular, CFI does not require bug-free programs which is very hard to achieve in practice given the fact that many programs are still being written in type-unsafe languages (like C/C++). On the other hand, existing CFI solutions suffer from performance problems or do a trade-off between performance and security leading to insecure and coarse-grained CFI policies. Our goal in this paper is to present a new hardware-assisted CFI framework that enforces fine-grained CFI policies using a per-function label approach and a state model where active labels (i.e., currently executing functions) are maintained in a dedicated memory area. We also outlined our implementation plan and elaborated on challenges. We believe that CFI in embedded systems is an enabling technology for other security services such as runtime attestation of embedded systems.

8. REFERENCES

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

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*, S&P '08, 2008.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1996.

[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] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Network and Distributed System Security Symposium*, NDSS '12, 2012.

[7] J. DeMott. Bypassing EMET 4.1. `http://labs.bromium.com/2014/02/24/bypassing-emet-4-1/`, 2014.

[8] 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.

[9] T. H. Jannik Pewny. Control-flow restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference*, ACSAC '13, 2013.

[10] A. K. Kanuparthi, J. Rajendran, M. Zahran, and R. Karri. Dynamic sequence checking of programs to detect code reuse attacks. Technical report, 2013. `http://isis.poly.edu/~arun/tvlsi.pdf`.

[11] P. Kocher, R. Lee, G. McGraw, and A. Raghunathan. Security as a new dimension in embedded system design. In *Annual Design Automation Conference*, DAC '04, 2004.

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

[13] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4), July 2004.

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

[15] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embed. Comput. Syst.*, 3(3), Aug. 2004.

[16] 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.

[17] 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 *IEEE Symposium on Security and Privacy*, S&P '13, 2013.

[18] 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.

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

[20] T. Zhang, X. Zhuang, S. Pande, and W. Lee. Anomalous path detection with hardware support. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '05, 2005.