A FPGA-based control-flow integrity solution for securing bare-metal embedded systems

A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems

Nicolò Maunero*, Paolo Prinetto*, Gianluca Roascio*, Antonio Varriale†
*Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy
Cybersecurity National Laboratory, Consorzio Interuniversitario Nazionale per l'Informatica (CINI)
{nicolo.maunero, paolo.prinetto, gianluca.roascio}@polito.it
†Blu5 Labs Ltd., Malta
av@blu5labs.eu

***Abstract*—Memory corruption vulnerabilities, mainly present in C and C++ applications, may enable attackers to maliciously take control over the program running on a target machine by forcing it to execute an unintended sequence of instructions present in memory. This is the principle of modern Code-Reuse Attacks (CRAs) and of famous attack paradigms as Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP). Control-Flow Integrity (CFI) is a promising approach to protect against such runtime attacks. Recently, many CFI-based solutions have been proposed, resorting to both hardware and software implementations. However, many of these solutions are hardly applicable to microcontroller systems, often very resource-limited. The paper presents a generic, portable, and lightweight CFI solution for bare-metal embedded systems, i.e., systems that execute firmware directly from their Flash memory, without any Operating System. The proposed defense mixes software and hardware instrumentation and is based on monitoring the Control-Flow Graph (CFG) with an FPGA connected to the CPU. The solution, applicable in principle to any architecture which disposes of an FPGA, forces all control-flow transfers to be compliant with the CFG, and preserves the execution context from possible corruption when entering unpredictable code such as Interrupt Services Routines (ISR).**

***Index Terms*—security, code-reuse attacks, return-oriented programming, ROP, JOP, embedded systems, microcontrollers, firmware, bare-metal, backward edges, forward edges, interrupt**

I. Introduction

Embedded devices are nowadays playing a central role in our lives, as they control most of the objects surrounding us. In addition, such systems create a network of connections that goes far beyond simple isolated LANs and links up devices all over the world. A huge amount of sensitive data is thus exchanged, and related security and privacy issues must be addressed.

In addition to communication security, a relevant aspect is the protection of devices themselves and their resilience to unauthorised intrusions. Physical security is certainly a first step, but not enough, since vulnerabilities may be contained in the code that the systems execute. Many of these vulnerabilities derive from the widespread use of very powerful languages such as C and C++. These languages guarantee a high degree of low-level control, but at the same time they allow programmers to freely manipulate memory pointers, so that common weaknesses such as *buffer overflow* [1] or *dangling pointers* [3] come out.

These vulnerabilities open the door to a family of exploits commonly known as *Code-Reuse Attacks (CRA)*, in which the flow of the program is redirected to portions of code already present in memory but not intended to be executed in that order. *Return-Oriented Programming (ROP)* [44] [13] [40] and *Jump-Oriented Programming (JOP)* [10] [18] are attack paradigms belonging to this category. In a paper of 2005 by Abadi *et al.* [7], *Control-Flow Integrity (CFI)* was suggested as a basic defence approach. CFI states that every control-flow transfer occurring during the execution of a program must target a valid destination, as stated in its *Control-Flow Graph (CFG)* computed ahead of time. Basically, the program behaviour is observed by an online monitor (software or hardware), which is able to ensure that no transfer happens out of those established in its CFG.

In literature, several implementations of CFI have been presented. Purely software solutions are mostly based on code instrumentation [9] [17], with additional checks on the destination of the control-flow transfers. These methods can however result in a considerable overhead in terms of added instructions, allocated data structures and/or execution times, often not acceptable for real-time systems with limited resources. In other cases, solutions based on multitasking have been proposed [22] [31] [54], very modular but inapplicable when the code is directly executed by the processor without the intervention of an Operating System (*bare-metal machines*).

Hardware-based CFI solutions [50] [23] [20] try to overcome these limits by proposing CFI monitors directly installed at hardware level. The program executes without spending time for checks, which are performed almost transparently in a parallel and much faster way. Moreover, sensitive information are not even visible by the main execution, which cannot access it in any way.

If hardware-based CFI is advantageous for these reasons, on the other hand, providing support for a hardware unit that directly accesses pipeline registers [25] or the bus between the core and the instruction cache [21] becomes unaffordable, as it is necessary to modify the internal design of the microcontrollers. This can be avoided if the hardware is equipped with a reconfigurable component, e.g., an FPGA, which can be

used to implement a CFI monitor without touching the internal architecture of the processors.

The goal of the present work is therefore to propose a solution for bare-metal microcontroller systems that exploits the presence of a FPGA onto which the CFI monitor can be synthesised. The solution lies halfway between software techniques and hardware techniques, with a minimal binary instrumentation based on single **`write`** machine instructions: these are used to communicate to the external reprogrammable device the information about the status of the CFG. The monitor validates the information received and stops the processor activity when a deviation is detected, via a security violation hard fault. The solution is in principle applicable to any architecture provided with an FPGA, and does not involve modifications to internal structure of the processors.

Outline. In the following of this Section, we offer a brief digression on what is the CPU-FPGA cooperation trend, to better contextualize our work; Section II provides some technical background on Control-Flow Hijacking attacks; Section III presents main state-of-the-art hardware-based CFI solutions; Section IV motivates our work and lists the challenges that are addressed; Section V presents our FPGA-based solution; Section VI lists the experimental results obtained from a preliminary implementation; Section VII finally concludes the paper.

A. The CPU-FPGA cooperation

According to latest Gartner research about the future of Infrastructure and Operations [6], FPGA will be part of the top 10 technologies to drive innovation through 2024.

The most recent strategies depict a primary interest of using FPGA in server-side hybrid chips. Nevertheless, the rise of 5G technology, the consequently spread of IoT and OT infrastructures, and the need for real time insights and localised actions, are forcing to deploy edge-computing solutions to process data closer to the source of generation. It is expected that, over the next few years, hardware vendors will focus on delivering computing hardware to execute complex, compute-intensive functions at the edge. In this context, hybrid chips based on CPU and FPGA components, will be the easiest and most power/cost-effective way to meet the new edge computing hardware requirements. Although there are still a few examples on the market, mostly provided by FPGA vendors who embed ARM or NIOS cores in their devices, hybrid CPU+FPGA chips are expected to become increasingly popular in the next years.

FPGA and CPU devices are already employed in many projects as separate components interconnected through a parallel bus and mounted on the same electronic board. In most of the cases, the FPGA is mapped as a memory device whereas the CPU acts as a master of the system. However, the mobile terminals market is driving a new trend, which aims to replacing the parallel bus with serial *differential lines* in order to reduce the final device size and, at the same time, to increase the data transfer rate. In the past decade, this process already happened in the PC world, when the PCI parallel bus was replaced by the PCIe differential lanes based bus. In terms of architectural access, we are talking of a migration from *memory-mapped* devices to *port-mapped* devices.

In any case, since the new serial buses affect the memory components, it is expected that the CPU will adapt the instruction set to atomically manage the memory access with a *single-instruction paradigm*, either mapping the **`LOAD`**/**`STORE`** instructions to the new serial buses or introducing **`IN`**/**`OUT`** instructions to manage the serial memory access.

II. Background

The IEEE Spectrum ranking of top programming languages [4] reports C and C++ as respectively 2nd and 3rd most used languages in the embedded system domain still in 2019. The reasons may be many, but there is no doubt that one of the great advantages in their use is the availability of low-level control structure that allows a deep optimisation in resource usage without losing the advantages of high-level statements. Although, the direct management of data structures in memory and the free manipulation of pointers originate a large number of vulnerabilities. The lack of memory safety capabilities (such as a strong typization, present in other modern languages) enables attackers to exploit these bugs by maliciously altering the program's behaviour.

One of the most famous vulnerabilities of this kind is *buffer overflow* [1], which is caused by incrementing or decrementing a pointer without proper boundary checks. This may result in out-of-bounds writes which corrupt adjacent data on stack, heap or other zones. Similar problems may rise when *indexing bugs* are present in the code, i.e., when boundary checks over an index for a given data structure are missing or incomplete. Indexing bugs derive from programming errors collectively known as *integer-related errors*, such as *integer overflow* [2], incorrect signedness or wrong pointer casting.

Famous are also *use-after-free* vulnerabilities [3], for which a pointer is mistakenly used after the area it points has been freed and released to the memory management system. After the free, the pointer still points to the deallocated region, which in the meanwhile can have been written with other data. The consequence is that newly allocated data in the heap may be corrupted by accessing it by these *dangling* pointers.

Memory vulnerabilities described above may enable attackers to maliciously take control over the program by forcing it to execute an unintended sequence of instructions. This exploit is generally called *Arbitrary Code Execution* (ACE). To achieve ACE, attackers tamper with the instruction pointer, which in most architectures is referred to as *Program Counter* (`PC`). The `PC` stores the address of the next instruction to be executed: being able to control its content means being able to decide the next instruction to be executed.

The control over the instruction pointer can be taken, for instance, by corrupting the memory operand of an instruction that copies that value into the `PC` (*indirect control-flow transfer instructions*). **`RET`** and some formats of **`CALL`** and **`JMP`** are example of such instructions, but, in general, any instruction

that treats the PC register as a destination register for a computing operation can be exploited.

The PC value is corrupted to point to the attacker's *payload*. This was traditionally injected together with the corrupted instruction pointer into the program memory program (*Code Injection*) thank to stack memory vulnerabilities [38]. Such exploits were made practically impossible after the wide adoption of main architectural countermeasures like *Data Execution Prevention* (DEP) [47] and *Write XOR Execute* policy [48], for which a memory location cannot be both writable (W) and executable (X) at runtime. Attackers then devised a new attack paradigm, in which the payload is composed of snippets of code already present in the program memory, but not meant to be executed in that order. This was how *Code Reuse Attacks* (CRA) were born. In a paper of 2007 by Shacham *et al.* [44], the authors theorized that "*in any sufficiently large body of executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able [...] to cause the exploited program to undertake arbitrary computation*". The control flow can be diverted to execute a series of small sequences of instructions, each ending with an indirect control-flow transfer instruction, known as *gadgets*. In large codebases present in every C application, such as libc, the amount of gadgets that can be extracted is high, and the attackers achieve the maximum of expressiveness [49].

This is the basic idea behind a famous attack paradigm known as *Return-Oriented Programming* (ROP) [40]. In ROP, the attackers write their malicious code using, instead of instructions, the gadgets found in the code of the system to be attacked as basic "bricks". These gadgets may perform any kind of general-purpose action, as copying values from registers to others, loading values from memory, or performing arithmetic and/or logic operations. The common property they must have is that their last instruction must always be a **RET** instruction. Once identified the set of gadgets, attackers fill the stack with a list of fake return addresses exploiting a memory vulnerability (Figure 1). Each of the injected addresses points to the beginning of each of the identified gadgets.

The attack starts when the function that contains the vulnerability returns: by executing the **RET**, the processor copies the first corrupted value into the PC, and the program flow is redirected to the first gadget of the sequence. Once the first gadget is finished, another **RET** is executed, that carries the flow to the second gadget, then to the third, and so on (Figure 2).

ROP was demonstrated to be effective over many different architectures [13] [27] [16] [15] [33], and then the concept was extended to non-**RET**-ended gadgets. Indirect formats of **JMP** and **CALL** can be used as well to reach instructions at will. Concepts like *Jump-Oriented Programming* (JOP) [10] [18], *Call-Oriented Programming* (COP) [42], and others [43] [30] were introduced.

III. Related Work

Literature has been enriched with a considerable amount of CFI solutions, ranging from purely software implementations

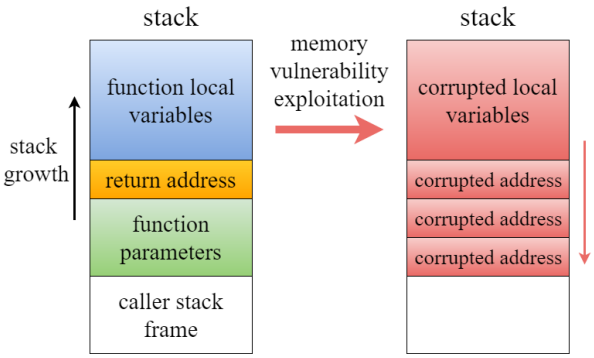


Fig. 1: A ROP attack starts filling the stack with a list of fake return addresses.

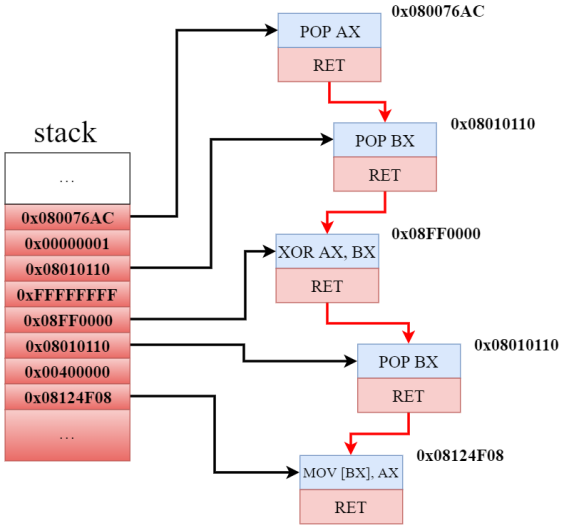


Fig. 2: An example of a ROP attack.

[7] [9] [17], to techniques that take advantage of features made available by Operating Systems [22] [31] [32] [54], to hardware-based solutions. Since this last is the field of our proposed technique, an overview of the most significant examples is here listed. The various techniques offered can be classified into *families*.

Branch target or instruction protection. One way to prevent code-redirection attacks is to make indirect branches operations protected by a key, which the external attacker does not know. The authors of [39] propose to insert a module in the architecture that automatically encrypts the routine return address before pushing it onto the stack at the call time, and that decrypts it when the **RET** is executed. Such an on-the-fly pointer encryption/decryption mechanism is also presented in [35]. In [36], a slightly different approach is instead adopted, which involves the encryption not of the addresses but of the indirect jump target instructions. This encryption is done at load-time, when the code is loaded in memory. At runtime, every time an indirect branch is performed, the processor

automatically decrypts the target instruction, and if an invalid instruction results from the decryption (i.e., the attacker redirected the branch elsewhere), the execution stops. The encryption is done using a key extracted from a processor PUF [29], so that it does not have to be stored but it is generated every time). Others present solution to protect code pointers at higher level of abstraction, e.g., marking code memory pointers as compile-time-generated or run-time-generated [19], or saving them in a separate stack through dedicate instructions to isolate them from memory errors [28] [41].

Shadow Call Stack (SCS). In [8], [12] and [24], the authors augment RISC-V soft cores with a shadow stack, i.e., a hidden and duplicated stack with respect to the one used by the program, that stores routine return addresses. At return time, the content of the shadow stack is authentic, so if a mismatch with the original stack is found, the attack is discovered and made ineffective.

Basic Block hashing. This technique is based on the progressive computation at runtime of the hash of the executed instructions, to check whether they are compliant to what has been pre-computed before execution. In [50], this task is performed by a module tightly connected to the processor that has direct access to the program counter and the instruction register. In [25], the checking modules are directly inserted into the pipeline stages. The authors of [21] and [14] instead propose validation modules placed halfway between the instruction cache and the processor, to sniff the instruction flow on the bus.

Modification of Branch Predition circuitry. Branch buffering and branch prediction modules, present in modern processors, can be enabled to perform security tasks, as presented in [45], [53] or [34].

Insertion of security features in the Instruction Set Architecture (ISA). The instruction set of a processor architecture is expanded to support instructions to check for CFI, which can be inserted by the programmer to protect branches. The core is therefore augmented with internal data structures as label stacks to protect backward edges and label registers to protect forward edges. As a matter of example, the works in [23], [20] and [46] presented an implementation of this paradigm on soft processors of the SPARC family.

IV. Challenges

All the defense mechanisms presented above are certainly getting the point, i.e., going down to the lowest possible level to set up the defense, which makes the system more resilient to whatever happens on top of it. However, these solutions are highly invasive from an architectural point of view, since they require the internal design of the processors to support these security features. Such an approach leads to higher costs (to produce custom chips), low modularity and low upgradeability, features which are instead achieved if the defense is implemented on a piece of reconfigurable hardware connected or included in the processor.

In light of the above, from our point of view it is important to page a solution that:

- aims at protecting microcontroller-based systems even when they directly execute a firmware stored in the Flash memory without the support of an Operating System (*bare-metal*), being thus independent of the facilities offered by OS's, such as multitasking or privileged execution levels;
- exploits the advantages of a hardware-based defense applicable without designing custom microcontrollers to have a protection, by mixing binary instrumentation techniques and low-level runtime monitoring based on reprogrammable hardware (FPGA);
- sets up an efficient defense mechanism that does not rely on secrets of any kind (e.g., encryption keys or secure identifiers) to be hidden by memory protection mechanisms or similar;
- cares about the strict requirements that these systems have in terms of resource occupation and execution times, and therefore aims at minimally impacting the system configuration and behavior, by properly selecting the edges to be protected;
- takes into consideration the problem of hardware interrupts, as explained in [37]: if not properly protected, the context of the program, including sensitive elements from the CFI point of view, can be corrupted with consequent loss of effectiveness of the solution.

V. Our Approach

The proposed solution aims at ensuring that (i) all branches target a valid location, (ii) the program context be not corrupted during sudden calls to Interrupt Service Routines (ISRs). The implemented CFI monitor is a module synthesised on a FPGA connected to the CPU it via a serial or parallel interface. An instrumented version of the program runs on the CPU and awakes the monitor by sending *sensitive data* about branches and context. In parallel, without stopping the processor activity, the monitor processes these data and interrupts the CPU only if they are not compliant with the expected ones. The CFI monitor is the only IP present on the reconfigurable hardware device. The cooperation system between CPU and FPGA is depicted in Figure 3.

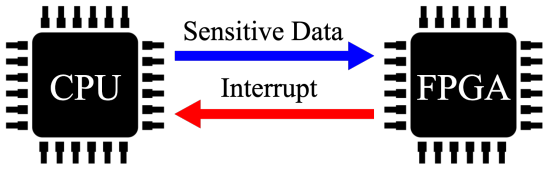


Fig. 3: The CPU-FPGA cooperation system for protection.

The program is instrumented so that single **`OUT/STORE`** instructions (called hereinafter **`write`** instructions for simplicity) are added in specific points of the code to communicate to the monitor two kinds of data:

- *labels* to uniquely identify a position within the code (for edge protection);

- *values* contained in specific registers (for context protection).

Together with data, the CPU must also communicate an *opcode*, that distinghuishes the kind of the provided data and instruct the monitor on the the right operation to be performed.

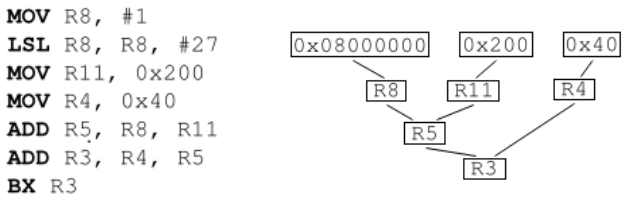
The sequel of this Section is organised as follows: we first introduce a classification of the CFG edges to define those needing protection. Then, the problem of context corruption and why context protection is needed are explained. The two phases of the protection (*online* and *offline*) are eventually presented, followed by some remarks about the architecture and the actual implementation of the proposed solution.

Classification and Identification of Edges

As already mentioned, the CFG is the set of connections between the basic blocks (BB) of the program through *edges* that correspond to control-flow transfers. Edges can be classified depending on the transfer instruction that generates them. They can be first distinguished in *forward edges* and *backward edges*, where the latter are edges connecting a BB to another which immediately follows (in terms of static position within the code) a block visited previously. These are typically the return edges from a routine. "Forward edges" refers to all the other edges that connect a BB to another elsewhere in the code. In most cases, these are the calling edges of a routine, but they can also be jumping edges within a same routine.

We refer as *target* of an edge to the BB pointed by that edge. From this definition, we can define *direct edges* and *indirect edges*. Direct edges are edges whose target is expressed as a label encoded within the instruction itself, while indirect edges are edges whose target is expressed by the value of a program data.

An *origin tree* of an edge target is a tree whose root is the location (register or memory address) used as argument of the instruction generating the edge, and which traces all the locations used to compose the value of the target up to the origin. Figure 4 shows a snippet of code in ARM-Assembly-like language ending with the edge-generating instruction **BX** R3 (indirect jump to address stored in R3), with the relative origin tree for R3. For direct edges, the origin tree



(a) (b)

Fig. 4: Snippet of code in ARM-Assembly-like language: (a) Code (b) The "origin tree" for R3.

is a trivial tree composed of a root node, only. For indirect edges, since the target is a program data, the tree can instead be complex at will. However, in bare-metal embedded systems, we can assume the entire code is already available in a single binary stored in Flash, and there are no modules linked at runtime. We can also assume that the code remains constant during activity. The result is that *the construction of the origin tree is always possible*, no matter the complexity in constructing it. This represents a key point for the proposed protection mechanism.

If the origin tree is always entirely reconstructable, then it is possible to list it all, from the root to the leaves. The leaves of this tree will be values that cannot further derive from other locations, i.e., they are either constant values or inputs taken from the outside. Assuming that an external input can never be used to compose a code pointer (because even in the case of a switch-case statement over an input, there is always a translation in a readable or predictable constant value, which then becomes the leaf), or in alternative we impose it as a design rule, then *the set of targets of an edge is always finite and enumerable*, and that set is a strict subset of all possible code locations. In direct edge case, the cardinality of this set will be 1, while it will be greater than or equal to 1 in indirect edge case. It follows (and *this* is the point) that under these assumptions *it is always possible to list all the destinations of all the edges of a CFG*, and thus, it is always possible to completely protect the integrity of the control flow.

Introduced all these definitions, it is possible finally to divide edges into *insecure edges* and *secure edges*, i.e., edges that need protection against control-flow hijacking and edges that need not. This is mainly important to reduce the number of code areas to be protected, primary target for embedded systems with limited potential.

We assume as insecure an edge whose target has an origin tree that contains at least one node in an area at risk of corruption, i.e., the data memory (if we consider the code memory incorruptible). In other words, no matter which are the leaves of the origin tree of its target, an edge is insecure when its target is even partially composed with data coming from data memory. This immediately implies that all direct edges are secure, but also all indirect edges composed with values that never *exit the code* (intended as union of code memory and processor registers) to go in the data memory.

This approach can be considered as conservative (think of the case in which a value is saved in memory and retrieved few instructions later). To prevent the creation of this kind of false positives, it would be necessary to go further in the analysis of the code, to investigate about the actual possibility of corruption between the store and the load instructions. However, this would mean taking into consideration a memory vulnerability database, and even looking only for the vulnerabilities known so far, this would be not trivial, and moreover, the unknown vulnerabilities would not be taken into account.

In conclusion, if the edge is insecure, then it must be instrumented so that a CFI monitor, at runtime, is able to decide whether it is actually pointing to one of its valid targets. In the case of an insecure forward edge, there is no way to say which of these points is the right one, because this depends on the execution, so the monitor can do nothing but ensuring

that all valid target can be reached. In the case of an insecure backward edge, the monitor can instead enforce a single target, because in addition to store all the possible destinations, it is also possible to store in the monitor the identifier of the BB to be executed at return time, and so the execution is forced to go back there.

Interrupt Service Routines

The assumptions made so far are valid only if one does not consider that the processor, in undefined moments of the execution, can jump to execute special routines to serve interrupt requests (Interrupt Service Routines, ISRs). As explained in [37], there is no static analysis that can forecast in which order or where in the code these routines will be called, so they can never be part of a predefined CFG. Yet, the ISRs are full-fledged routines, which operate on data and registers and which preserve the current program status moving it into memory. The result is that the origin tree that can be constructed from a static analysis as we have seen so far become invalid.

To preserve what has been assumed up to now, it must be therefore ensured that the execution context when entering into an ISR will be equal to the one when resuming the main program. To achieve this, an additional specific instrumentation is needed, based on the validation of the registers' content, with a double check before and after the execution of the service routine.

Protection Mechanism

As any other, our CFI solution resorts to an *offline* phase and an *online* phase as well.

In the offline phase, the firmware to be protected is first compiled, then a static analysis identifies different categories of *critical points* in the Assembly code. Critical points are locations within the code that require the monitor intervention for control-flow verification in the online phase. In correspondence of such points, some data must therefore be sent to the FPGA, i.e., a **`write`** instruction must be inserted. For each BB that contains a critical point, a *unique identifier* is produced and inserted into the code as a constant. The inserted **`write`** will therefore send the identifier of the BB, using as address a code to instruct the monitor. For edge protection, seven categories of critical points are identified:

1) *Forward insecure edges with single target*: the ID of the source BB is sent to the monitor before the transfer, and the ID of the target BB is sent after the transfer. Internally, the monitor combines the two IDs, and if the edge is valid, the execution can proceed, otherwise the CPU activity is immediately interrupted via a security fault using the interrupt line;
2) *Backward insecure edges with single target*: same as above;
3) *Forward insecure edges with multiple targets*: same as the case of single target, but here all target locations are instrumented;
4) *Forward secure edge to a routine ending with a backward insecure edge with multiple targets*: this transfer is not to be protected, but the ID of the BB to which the called routine must return is sent. In the monitor, the ID is pushed on top of a stack structure;
5) *Backward insecure edges with multiple targets*: same as 2), but the ID of the target BB must correspond to the ID sent as described in 4). In this regard, the top of the stack is popped and compared to the ID of the target. If a mismatch is found, the violation fault is triggered;
6) *Forward insecure edge to a routine ending with a backward insecure edge with single target*: again, as in 4), the return BB ID is sent, but also the ID of the target BB is sent after the transfer (to verify both caller identity at return time and validity of destination of the present call);
7) *Forward insecure edge to a routine ending with a backward insecure edge with multiple targets*: same as above, but here all possible return sites are instrumented;

For context protection, two categories of critical points are identified:

1) *Entry point of an Interrupt Service Routine (ISR)*: a given number of consecutive **`write`**s are inserted as first instructions of the ISR, storing the content of registers which, upon entering an ISR, are automatically pushed by the processor architecture (e.g., in case of ARM, `R0`, `R1`, `R2`, `R3`, `R12`, `LR`, `PC` and the status register `xPSR`), plus the registers which are additionally used by that ISR. Internally, the monitor saves all these values on top of a dedicated stack structure;
2) *Exit point of an Interrupt Service Routine (ISR)*: before leaving, the same number of **`write`**s performed at the entry point for the same registers, are performed in reverse order. The program transfers from the top of its stack to the monitor, which compares the received values with the ones on top of its own dedicated stack. If a mismatch is found, a violation is notified through the interrupt line.

After the instrumentation process described above, two items are available:

1) the instrumented executable binary;
2) a table containing all the instrumented edges, intended as a set of ID pairs (source BB, target BB).

The edge table is converted into a memory initialisation file (.mif) which is then used to produce a read-only memory (ROM) block to be placed inside the monitor architecture. The RT-level description of the monitor is synthesised into a *bitstream* used to program the FPGA.

Once all the sources are ready, as last step of the offline phase, the programming part takes place: a *secure boot loader* both loads the instrumented version of the firmware and programs the FPGA, correctly setting the CPU-FPGA interface in order to allow the runtime interaction. The online phase now starts, with the FPGA acting as a monitor in response

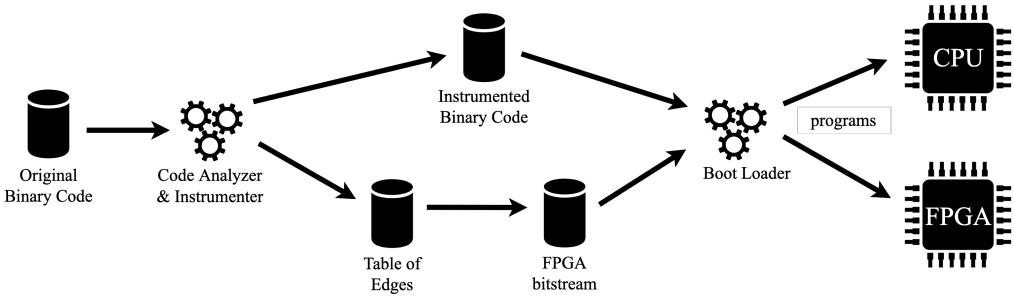


Fig. 5: The workflow of the analysis and instrumentation process.

to control-flow information received by the instrumented program.

In their communication, CPU and FPGA do not need to establish synchronization, as they share the same oscillator for clock signal. Possibly, they may run at different frequencies, multiplying or dividing the oscillator frequency. Anyway, this is decided once for all during configuration, and both actors are aware of the relative speed they have.

The workflow of the analysis and instrumentation process is presented in Figure 5.

Monitor Internal Structure

In summary, the monitor relies on three different data structures:

- an *edge table* which encodes the information about all consented control-flow transfers, as pairs of source BB ID and target BB ID;
- a *secure ID stack*, where it pushes the identifiers received to protect backward insecure edges with multiple destinations;
- a *secure register stack*, where it pushes the context of the program upon entering an ISR and checks whether this has remained the same or has been corrupted upon exiting the ISR.

A central *control and check unit* decodes the commands coming from the CPU to generate consequent reads and writes on these three storage blocks, as well as it verifies, through a set of comparators, that the received data are the expected ones. As an output, the unit controls the *interrupt line*, which notifies the CPU that an attempt to redirect the control flow is in progress.

The unit also contains a *timer*, crucial for security. In fact, when protecting an edge, a very stringent timeout must be triggered as soon as the source ID is received. To jump to any gadget in memory, the attacker must pass through one of the instrumented zones, because there is no trampoline which remains unprotected after the instrumentation. When it succeed in tampering with the branch target and jumps to his payload, there is no instrumentation in that position, unless the jump is compliant with the CFG (but when an attack is performed, this is not the case). Therefore, the monitor assumes an attack when, at timeout, the ID has not yet been received. Since CPU and FPGA share the same clock source, the length of the timeout is just the time for the execution of a branch, plus the time required to complete the **`OUT/STORE`** instruction, possibly multiplied or divided according to the relative CPU-FPGA frequency.

The impossibility to access the FPGA in the normal execution is set as a design rule to guarantee protection: the FPGA is considered as a private resource unusable by the program, so any possible read or write from/to the FPGA is removed during the offline phase, in such a way that no accesses other than those provided by the protection are consented.

The overall block diagram of the CFI monitor is depicted in Figure 6.

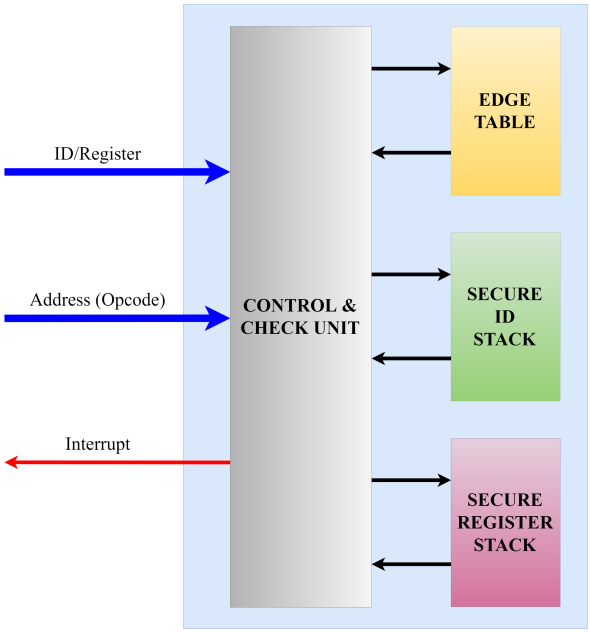


Fig. 6: CFI monitor block diagram.

Involved Overhead

As shown, in terms of code equipment, the defense is implemented simply by performing **`write`** instructions into the external device. Thus, the need to allocate memory to store CFG information is overcome, as well as it is eliminated

the computational overhead necessary for validity checks. Conceptually, the **write** instructions required are:

- just 1 for each instrumented location for edge protection;
- n for each instrumented location for context protection, where n is the number of registers pushed by default by the architecture upon entering an ISR, plus the registers pushed because used by the routine.

The term *conceptually* is here a keyword, because to reach exactly 1 and n **write** in each case, the architecture has to support specific features. In particular, additional machine instructions are needed when (i) the ISA does not support writing immediate values to immediate addresses, (ii) mismatches are present in the width of the involved buses.

Concerning the hardware part of the defense, the overhead can be evaluated in terms of the amount of occupied area on the reprogrammable device. The proposed solution requires the CFI monitor be the only module in the FPGA. Required resources are mostly *memory* resources, for the edge ROM and the two stacks for IDs and registers. These blocks must be properly dimensioned to accommodate all the edges and the maximum amount of forecasted stackable IDs and registers. The additional logic, including the state machine, some comparator and some registers for intermediate data storage, occupies a marginal area, as shown in next Section.

In terms of timing, the FPGA computation needs to be completed in the shortest time possible, in order to inform the processor about an attack as soon as possible. To achieve this, an intelligent encoding for the consented edges is adopted, which allows to access the table with an $O(1)$ complexity (implementing it as a hash table) after a fast and lightweigth combination of source and target IDs.

Trading off Security and Complexity

The features which may limit the feasibility of our solution are:

1) a too large execution time overhead due to the added instructions, so that it is no longer possible to meet some real-time constraints;
2) too much latency between the **write** of the sensitive data and the attack detection, so that the attacker can jump to dangerous code and perform destructive actions in that time window.

Both problems can be faced by trading off security and performances. In particular, the former problem can be tackled by the system designer, who could resort to a "partial" protection: insecure edges belonging to paths proven to be "critical" from the performance point of view could be left "unprotected". This could be justified with a deeper analysis of code vulnerabilities or simply by assuming the risk of such a choice.

To address the latter problem, the designer should identify the code sections that, within the response time window of the monitor, may cause irreparable damage to the system functioning. These depend not only on the code, but also on the time the adopted architecture takes for executing it. If these dangerous sections are found, either the code is rewritten, so that it become harmless, or the relative frequency between CPU and FPGA should be properly tuned, so that the monitor is faster than an attacker.

VI. Experimental Results

In this Section, some preliminary experimental results deriving from the implementation and testing of our solution on a real device are presented. For the evaluation, the SE*cube*™ Chip [5] by Blu5 Group® has been used. SE*cube*™ is an open security-oriented platform, implemented as a 3D SiP (System-in-Package) integrating three components:

- A STM32F4 microcontroller by STMicroelectronics™ , embedding a ARM Cortex-M4 core, 2 MB of Flash and 256 KB of SRAM;
- A MachX02 FPGA by Lattice Semiconductor™ , hosting 7000 4-bit look-up tables and 240 Kbits of embedded SRAM;
- An EAL5+ certified Smart Card;

The chip was designed as a secure processor that acts as a slave to offer a master (which can be the main processor of a smartphone or of a PC) cryptographic and secure storage functionalities. In this regard, it was designed to resist the most common physical attacks [26] [11]. SE*cube*™ was not chosen only for the presence of the ST microcontroller and the FPGA, but also because this type of embedded processors, given the uses that are usually made of it, can be the victim of code redirection attacks, as shown in [51] and [52].

Our solution has been tested on some specific benchmarks for embedded devices, made available by the MiBench platform[1]. On the website, there are several archives containing the source code to be compiled on ARM platforms. We chose a set of 5 applications, and once obtained the binary, we performed the offline analysis and instrumentation process described in the previous Section. As a wrapper around the actual code, we implemented functions to start and stop the hardware timer present in the microcontroller, for measuring the execution times before and after the instrumentation.

The physical implementation of the SE*cube*™ platform and the STM32F4 architecture required increasing the number of machine instructions for each **write**. As an example, the external parallel interface has a 16-bit data bus, so two accesses are required to send 32-bit values. In addition, the **STR** machine instruction does not support an immediate address, so this must be first copied into a register.

On the FPGA side, we were able to implement a version of the monitor with 1024 entries for each of the two stacks and 8192 entries for the ROM edge table. These dimensions were decided statically before the benchmarking process, and anyway they are much more than needed for hosting critical information for each of the analyzed applications. As expected, we got an occupancy of 156 Kbits for the embedded FPGA memory ($\sim$69% of the total), which is to be attributed to the implementation of the three data structures. For the logic of our

[1] http://vhosts.eecs.umich.edu/mibench//

TABLE I: Preliminary Experimental Results

Benchmark	Inputs	Time (no prot.)	Time (prot.)	Overhead	# instr. (no prot.)	# instr. (prot.)	Overhead
SHA	Message of 100 KB	368449 μs	368457 μs	$< 0.01\%$	20453	21194	3.62%
RIJNDAEL	Message of 100 KB	1083568 μs	1083694 μs	0.01%	25471	26029	2.19%
DIJKSTRA	Matrix of 100x100 **int**	2880724 μs	2894391 μs	0.47%	20166	20912	3.70%
STRING	1331 strings (var. length)	178616 μs	180028 μs	0.79%	20060	20791	3.64%
BITCOUNT	12800 **int**	419545 μs	1233227 μs	193%	20192	20944	3.72%

monitor, we got 185 LUTs occupied ($\sim$3% of the total), which is perfectly expected for the simplicity of the implemented functionalities.

In Table I, we list data collected from the analysis of the benchmarks, in the form of execution times and number of instructions *before* and *after* the instrumentation process, and the involved overheads. We also list the dimension of the inputs given to each benchmark. In all experiments, the CPU was running at 180 MHz, while the FPGA at 90 MHz. On SE*cube*™ , CPU and FPGA share the same clock oscillator, so their synchronization is natural.

Looking at the Table, it is possible to notice the very low amount of additional code (always less than 4%). Results are instead conflicting as regards the execution times, with the first four examples at less than 1% overhead, while the last one at a very high overhead. This seems to contrast with the percentage of instructions added, which remains low as for the others. Actually, the discrepancy is generated because in the code there are very frequent indirect calls to functions consisting of a few instructions. The impact of the added code is therefore much greater than in other cases. This is also useful to show how the execution impact actually depends on how the code is architected. No solution that includes even a minimum of instrumentation can limit this, even if our solution greatly limits the percentage of total instructions added.

VII. Conclusions

In this paper, we presented a solution to guarantee the Control-Flow Integrity (CFI) of firmware running on bare-metal microcontrollers, which constitute a relevant part in the embedded domain. The work was mainly aimed at mitigating the drawbacks present in the previous state-of-the-art solutions. Using a mixture of binary instrumentation and hardware-based supervision, the solution entrusts the binary enforcement with the sole task of informing a CFI monitor present on an FPGA about the status of the CFG through simple additional **`write`** instructions at critical points. The hardware monitor is encharged of storing the information about the CFG and performing the needed computation for the validation. As demonstrated by experimental results, this technique greatly reduces the overhead of code necessary for protection. No multitasking is required, and the protection can be implemented on very simple systems and with minimal resources. In addition, the monitor is implemented on a reconfigurable hardware device, which frees the solution from need of designing custom CPU architectures to support the defense. The only constraint is the presence of a reconfigurable hardware, but as explained in Subsection I-A, this is a diffused market trend.

VIII. Acknowledgments

This paper is supported in part by European Union's Horizon 2020 research and innovation programme under grant agreement No. 830892, project SPARTA.

References

[1] CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer. https://cwe.mitre.org/data/definitions/119.html, 2019. [Online; accessed 28-October-2019].

[2] CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html, 2019. [Online; accessed 28-October-2019].

[3] CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html, 2019. [Online; accessed 28-October-2019].

[4] Interactive The Top Programming Languages 2019 - IEEE Spectrum. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019, 2019. [Online; accessed 28-October-2019].

[5] Multiple reconfigurable silicon in a single package. https://www.secube.eu, 2019. [Online; accessed 07-November-2019].

[6] Top 10 Technologies That Will Drive the Future of Infrastructure and Operations. https://www.gartner.com/en/documents/3970841/top-10-technologies-that-will-drive-the-future-of-infras, 2019. [Online; accessed 29-November-2019].

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

[8] M. Alam, D. Roy, S. Bhattacharya, V. Govindan, R.S. Chakraborty, and D. Mukhopadhyay. Smashclean: A hardware level mitigation to stack smashing attacks in openrisc. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–4. IEEE, 2016.

[9] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 353–362. ACM, 2011.

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

[11] M. Bollo, A. Carelli, S. Di Carlo, and P. Prinetto. Side-channel analysis of secube™ platform. In *2017 IEEE East-West Design Test Symposium (EWDTS)*, pages 1–5, Sep. 2017.

[12] C. Bresch, A. Michelet, L. Amato, T. Meyer, and D. Hely. A red team blue team approach towards a secure processor design with hardware shadow stack. In *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, pages 57–62, July 2017.

[13] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[14] A. Chaudhari and J. A. Abraham. Effective control flow integrity checks for intrusion detection. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pages 1–6, July 2018.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A.R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[16] S. Checkoway, A. J. Feldman, B. Kantor, J.A. Halderman, E. W. Felten, and H. Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. *EVT/WOTE*, 2009, 2009.

[17] L. Chen, J. Jiang, and D. Zhang. Code reuse prevention through control flow lazily check. In *2012 IEEE 18th Pacific Rim International Symposium on Dependable Computing*, pages 51–60, Nov 2012.

[18] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 20–29. ACM, 2011.

[19] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 378–387, June 2005.

[20] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis. Hcfi: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 38–49. ACM, 2016.

[21] J. Danger, A. Facon, S. Guilley, K. Heydemann, U. Kühne, A. Si Merabet, and M. Timbert. Ccfi-cache: A transparent and flexible hardware protection for code and control-flow integrity. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 529–536, Aug 2018.

[22] 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 *NDSS*, volume 26, pages 27–40, 2012.

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

[24] A. De, A. Basu, S. Ghosh, and T. Jaeger. Fixer: Flow integrity extensions for embedded risc-v. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 348–353, March 2019.

[25] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A. Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017.

[26] G. A. Farulla, A. J. Pane, P. Prinetto, and A. Varriale. An object-oriented open software architecture for security applications. In *2017 IEEE East-West Design Test Symposium (EWDTS)*, pages 1–6, Sep. 2017.

[27] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26. ACM, 2008.

[28] A. Francillon, D. Perito, and Claude C. Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 19–26. ACM, 2009.

[29] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160. ACM, 2002.

[30] Y. Guo, L. Chen, and G. Shi. Function-oriented programming: A new class of code reuse attack in c applications. In *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, May 2018.

[31] Z. Huang, T. Zheng, Y. Shi, and A. Li. A dynamic detection method against rop and jop. In *2012 International Conference on Systems and Informatics (ICSAI2012)*, pages 1072–1077, May 2012.

[32] Z. J. Huang, T. Zheng, and J. Liu. A dynamic detective method against rop attack on arm platform. In *2012 Second International Workshop on Software Engineering for Embedded Systems (SEES)*, pages 51–57, June 2012.

[33] T. Kornau et al. *Return oriented programming for the ARM architecture*. PhD thesis, Master's thesis, Ruhr-Universität Bochum, 2010.

[34] Y. Lee and G. Lee. Detecting code reuse attacks with branch prediction. *Computer*, 51(4):40–47, April 2018.

[35] Y. Lee and G. Lee. Hw-cdi: Hard-wired control data integrity. *IEEE Access*, 7:10811–10822, 2019.

[36] Y. Li, Z. Dai, and J. Li. A control flow integrity checking technique based on hardware support. In *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pages 2617–2621, Oct 2018.

[37] N. Maunero, P. Prinetto, and G. Roascio. Cfi: Control flow integrity or control flow interruption? In *2019 IEEE East-West Design Test Symposium (EWDTS)*, pages 1–6, Sep. 2019.

[38] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[39] P. Qiu, Y. Lyu, J. Zhang, D. Wang, and G. Qu. Control flow integrity based on lightweight encryption architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(7):1358–1369, July 2018.

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

[41] N. Roessler and A. DeHon. Protecting the stack with metadata policies and tagged hardware. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 478–495, May 2018.

[42] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. Pure-call oriented programming (pcop): chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques*, 14(2):139–156, May 2018.

[43] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, May 2015.

[44] H. Shacham et al. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security*, pages 552–561. New York,, 2007.

[45] Y. Shi and G. Lee. Augmenting branch predictor to secure program execution. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 10–19, June 2007.

[46] D. Sullivan, O. Arias, L. Davi, P. Larsen, A. Sadeghi, and Y. Jin. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.

[47] Microsoft Support. A detailed description of the Data Execution Prevention (DEP). https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in. [Online; accessed 28-October-2019].

[48] PaX Team. PaX Non-Executable Pages Design and Implementation. https://pax.grsecurity.net/docs/noexec.txt, 2003. [Online; accessed 28-October-2019].

[49] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *International Workshop on Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

[50] W. Wang, M. Liu, P. Du, Z. Zhao, Y. Tian, Q. Hao, and X. Wang. An architectural-enhanced secure embedded system with a novel hybrid search scheme. In *2017 International Conference on Software Security and Assurance (ICSSA)*, pages 116–120, July 2017.

[51] N. R. Weidler, D. Brown, S. A. Mitchel, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes. Return-oriented programming on a cortex-m processor. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 823–832, Aug 2017.

[52] N. R. Weidler, D. Brown, S. Mitchell, J. A. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes. Return-oriented programming on a resource constrained device. *Sustainable Computing: Informatics and Systems*, 22:244–256, 2019.

[53] Wenjian He, S. Das, W. Zhang, and Y. Liu. No-jump-into-basic-block: Enforce basic block cfi on the fly for real-world binaries. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017.

[54] Y. Xia, Y. Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, June 2012.