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

***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-based embedded 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 externally. The solution, also applicable to legacy systems, 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 that surround 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 put in motion, 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)* [54] [15] [50] and *Jump-Oriented Programming (JOP)* [11] [21] are attack paradigms belonging to this category. In a paper of 2005 by Abadi *et al.* [6], *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 [10] [19], 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 [27] [38] [62], very modular but inapplicable when the code is directly executed by the processor without the intervention of an Operating System (*bare-metal machines*).

Otherwise, hardware-based CFI solutions proposed to insert security features into the processor architecture, adding dedicated registers, instructions, and automatic control functions on the possible deviations of the program flow [28] [24] [56]. These techniques are very effective but very expensive as well, as they require redesigning the processor architecture, thus resulting inapplicable to legacy systems already operating in the field. Others have proposed the use of the debug interface provided by the processors to monitor the program behaviour with an external hardware block [37], which however leads to a prohibitive slowdown in performance.

The goal of the present work is therefore to propose a solution for bare-metal microcontroller systems that, for the reasons listed above, cannot directly adopt the approaches proposed so far. We want to stay halfway between software techniques and hardware techniques by presenting a generic and lightweight solution, which is based on monitoring the firmware run by a microcontroller by connecting a FPGA to an external standard parallel interface and by synthesizing the monitor onto it. Our technique uses a minimal binary instrumentation based on single `write` machine instructions to communicate to the external 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

suitable for any type of platform, future but also already in the field (*legacy*). Not being bound to any particular architectural feature, it does not require the modification of the internal structure of the microcontrollers and it just requires a parallel interface (present in any microcontroller) and the connection of an external FPGA (without the need to fabricate new silicon).

The rest of the paper is organised as follows: Section II provides some technical background on Control-Flow Hijacking attacks; Section III analyses more in details common solutions and their problems under certain conditions; Section IV motivates our work and lists the challenges that are addressed; Section VI presents our FPGA-based solution; Section VII lists the experimental results obtained from a preliminary implementation; Section VIII finally concludes the paper.

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 flaws 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 [47]. Such exploits were made practically impossible after the wide adoption of main architectural countermeasures like *Data Execution Prevention* (DEP) [57] and *Write XOR Execute* policy [58], 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.* [54], 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 [59].

This is the basic idea behind a famous attack paradigm known as *Return-Oriented Programming* (ROP) [50]. 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 theyr 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 [15] [33] [18] [17] [40], 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) [11] [21], *Call-Oriented Programming* (COP) [52], and others [53]

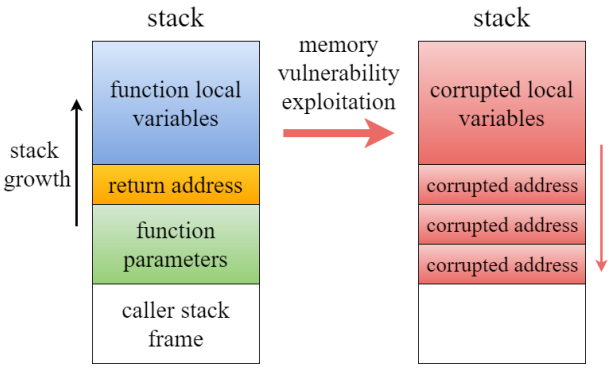


Fig. 1: A ROP attack starts filling the stack with a list of fake return addresses.

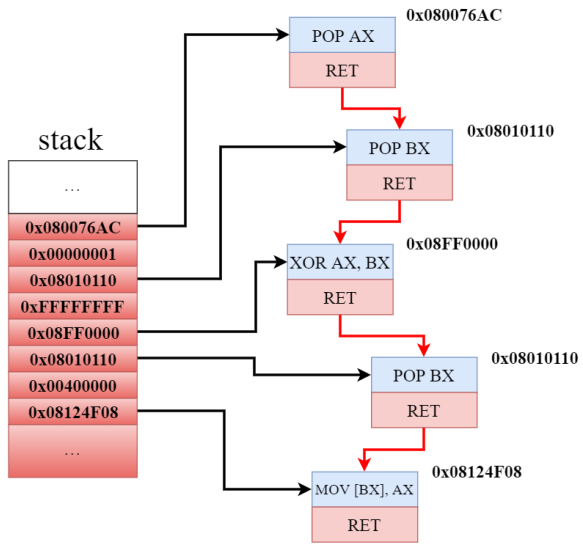


Fig. 2: An example of a ROP attack.

[36] were introduced.

III. Related Work

Over the years, the research community has tried to address the threat of CRA with solutions going in the direction of protecting the memory from its vulnerabilities. Randomisation of memory segments' location [9], protection of the return address on the stack [25] or replication of the entire stack for then validating it against corruption [34] [14] [13] and even heuristic techniques [20] [48] [23] have been presented.

In 2005 Abadi's article [6], however, a different approach to the problem started to be theorised. The paper introduced the concept of *Control-Flow Integrity* (CFI) for defending against CRA regardless of the vulnerability that may cause the exploit. The concept behind CFI is monitoring the program at runtime to detect abnormal diversion from what is stated in its *Control-Flow Graph* (CFG).

Referring to the definition given in [41], a CFG is a directed graph $G = (V, E)$ where V represents the set of the *basic blocks* of a program and E represents the set of *control-flow transfers* that connect such blocks. A basic block (BB) is defined as a linear sequence of program instructions having one entry (the first instruction executed) and no branches out except at the exit. All statements within a basic block are executed sequentially before transferring the control to the next basic block. In CFG domain, such control-flow transfers are usually referred to as *edges*.

The CFG is computed ahead of the execution, by statically analysing the source or the binary code of the program, or even by *profiling* all the possible paths through one or multiple test runs. Then, at runtime, no other control-flow transfers are allowed but those indicated in the CFG. Therefore, in almost all CFI solutions, it is possible to find (1) an *offline phase*, performed before runtime and often in a different environment from the one of the execution, mainly consisting of analysis and production of information needed later for verification, and (2) an *online phase*, performed at runtime and aimed at verifying that the parameters obtained by the offline phase are actually respected during the execution.

To ensure this, a *CFI monitor* is needed: an entity that, having been instructed by such parameters, is able to properly monitor the execution environment to get information about the performed flow transfers. In literature, plenty of different monitoring techniques have been presented, usually clustered into three categories: *OS-based*, *purely software-based* and *hardware-based*.

OS-based solutions

In OS-based solutions, the CFI monitor is the environment itself where the program runs, intended as a supervisor process that runs the supervised application within its own sandbox. This technique is referred to as *Dynamic Binary Instrumentation* (DBI), well explained in [30]: code is not executed directly, but rather analysed, instrumented, and compiled using a just-in-time (JIT) compiler. Analysis code relies on an API interface, and the DBI backend monitors program state switching between analysis routines and observed code execution. Solutions of this type are presented in [38] and, for ARM environments, in [39]. Alternatively to JIT compilation, a probe-based approach can be adopted, where the original program instructions are executed after having been patched with *trampolines* to analysis routines. Such a solution is presented in [27], implemented for iOS.

The paradigm followed in [62] is instead slightly different. Here, the presence of the Branch Table Store (BTS) module integrated in modern Intel cores [63] is exploited. The module fills a buffer in memory with the history of control-flow transfers. This buffer is in the memory space of a process which is parent of the process under supervision. The parent process contains all the information about the CFG of the child, and therefore it is able to block the attack and locate the anomaly.

All these solutions are highly portable, as they exploit utilities provided by Operating Systems. They undoubtedly have overhead in terms of occupied memory and execution

times, but being mostly designed for mobile or desktop OS's, the involved overheads are acceptable. Unfortunately, this is not the case in microcontroller-based systems, where not only resources are limited, but the firmware is often run directly from the Flash memory, without the intervention of any OS (bare-metal). Software solutions for protecting these systems must therefore integrate the CFI monitor into the supervised program itself.

Purely software-based solutions

The basic idea first introduced in Abadi's milestone paper [6] is to insert, within the binary code, additional checks before the transfer to verify its compliance with the CFG. A unique label is associated with each monitored basic block, and before an indirect transfer, it is checked whether the register or the memory location containing the destination address is actually corresponding to the destination label according to the CFG. If this is not the case, a violation is detected. In this way, the monitor is the program itself.

Another defense based on code instrumentation is *Control-Flow Locking* (CFL) [10], which inserts instructions to *lock* the execution before indirect transfers and instructions to *unlock* it at every valid target. The lock variable is set to a value which is verified by the unlock code before the execution be allowed to proceed. The lock code is also encharged to verify whether the ending basic block was unlocked and allowed to run, otherwise it notifies a violation. The authors of [10] propose to use a lock variable which only contains a 0 value for "unlocked" and a positive or negative value for identifying the type of branch. A CFL improvement is presented in [19], where the lock variable is split into several fields to contain also the identifier of the current BB and the BBs reachable from it.

Label-based and locking techniques reported so far can guarantee a very high level of protection, but in principle they are affected by the problem of overhead and of lack of isolation of sensitive variables, respectively. In the case of the classical label-based instrumentation, the number of instructions added to each branch site for checking can be high, which leads to an important extra occupation of the program memory and to execution times slowdown. In some cases, especially in the embedded domain, memory and timings are tightly constrained, and these solutions become very inconvenient. Moreover, in the case of the CFL, there is also the problem of protecting the lock variable somehow, totally isolating it from the rest of the execution environment. Finally, as shown in [45], serious problems may rise if the protection code is interrupted by a hardware interrupt. In such cases, the context of the program is entirely pushed onto the stack to free registers to execute the interrupt service routine (ISR). Supposing a vulnerable ISR is executed, the pushed context can be corrupted, and at resuming time the variables to be checked for security can be reloaded into registers with different values. The result is that the defense mechanism is escaped.

A recent paper by Nyman *et al.* [46] faces all these problems, proposing a solution for bare-metal embedded systems with a lower overhead in terms of occupied memory for instrumentation. The authors suggest replacing dangerous branches with calls to a single service routine (via a software interrupt), that runs inside the ARM TrustZone [8]. The TrustZone is by definition protected and runs at a higher privilege level compared to the application, in an isolated environment, so it can contain sensitive information about the CFG and can performs CFI checks in a secure manner. Though this is an excellent solution, it does not take into account the fact that there is literally a *jungle* of legacy bare-metal embedded systems out there, maybe not based on ARM architectures or not having any TrustZone.

Hardware-based solutions

An alternative way to solve overhead and isolation problems of a CFI monitor is to design it directly in hardware. In this way, the program runs normally and the CFI checks are performed much faster and almost transparently. Furthermore, the data structures containing CFG critical information belong to the monitor exclusively, and the main execution has no possibility of accessing it. Literature has indulged in this theme, providing a great variety of hardware-assisted CFI solutions. These can be grouped into *families*, among which we intend to present the most significant ones.

Branch target or instruction protection. The authors of [44] propose the encryption at load-time of the indirect branch target instructions, and the addition of a processor module that decrypts on-the-fly these addresses before loading them into the instruction register. The execution obviously crashes if a badly-decrypted instruction is run, and to mount a redirection attack the attacker must know the encryption key (extracted from a processor PUF [35], so generated each time and never stored). In [49], the authors propose a similar method which involves encrypting with a lightweight version of AES the return addresses at call time before pushing them, and decrypting them at return time. Other authors propose to solve the question at a higher level of abstraction, e.g., marking code memory pointers as compile-time-generated or run-time-generated [22], encrypting and decrypting them on-the-fly [43], or using special instructions for their load and store and placing them in a different special stack to isolate them from buffer overflow vulnerabilities [34] [51].

Shadow Call Stack (SCS). Authors in [7] [14] [29] propose the insertion of a call shadow stack into the architecture, implementing their solution on the RISC-V soft processor. In [29], the monitor is equipped with a table of the destinations allowed for each indirect transfer, and gets the status of the monitored program through a parallel interface with the main core which basically carries out the instruction register on a bus.

Basic Block hashing. This technique relies on calculating the allowed sequence of basic blocks before the execution and then verifying it at runtime by continuously computing the hash of the blocks. Authors in [60] propose as a hardware

trusted module parallel to the main processor that does this by reading the program counter and instruction register. The same is proposed in [31] by inserting checking modules directly connected to the pipeline stages. [26] and [16] propose to install validation modules between the processor and the instruction cache to sniff the instruction flow.

Modification of Branch Predition circuitry. Possible modifications or security extensions have also been studied for the branch buffering and prediction modules present in most processors [55] [61] [42].

Insertion of security features in the Instruction Set Architecture (ISA). These solutions give the programmer the possibility of inserting CFI-dedicated instructions in the program to be protected. The processor is therefore equipped with internal data structures as label stacks to protect backward edges and label registers to protect forward edges, and the instruction set is augmented with the opcodes necessary to manage them. Examples are the works presented in [28] [24] [56], where changes were made to the original design of some SPARC soft processors.

All these defense mechanisms 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. Rather than questioning their validity, we want to contest their *feasibility*. In fact, none of these solutions can be applied to a microcontroller that is already operating in the field, since each of these requires an even minimal hardware patch, only possible when a new version of the device is released. The authors of some of these above-mentioned works boast of not modifying the internal structure of the processors, although they know very well they are using a linguistic stratagem: inserting an additional module in the pipeline is not much different from installing a CFI verifier between the instruction cache and the core, because an intrusion is still required in the original design, which means *redesign*, even if the processor in itself is the same as before.

IV. Challenges

In light of the above and with respect to the solutions presented so far in literature for the Control-Flow Integrity and the their limits highlighted 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 having to wait for the updated version of the microcontroller to have a protection, making use for this purpose of a mixture of binary instrumentation techniques and low-level runtime monitoring based on external 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 (see Section VI);
- takes into consideration the problem of hardware interrupts and the fact that 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 (*interrupt awareness*).

V. Assumptions

In order for our solution to be effective, it is assumed that:

- the system is provided with a single-core processor that executes directly from the hardware a *single* application (hereinafter, generically referred to as *the program*) that does not include parallel computing, even theoretically, and therefore does not support any context switching;
- the entire code to be executed is already loaded in memory and cannot be modified in any way, neither from outside nor from inside;
- the microcontroller has a parallel interface to an external device mapped in the addressing space of the processor;
- the whole system is closed inside a tamper-proof package, so that it is not possible to inject faults or disconnect internal components unless making it unusable.

VI. 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 external to the MCU and connected to it via a standard parallel interface. An instrumented version of the program runs on the MCU and awakes the monitor by sending *sensitive data* about branches and context. The monitor acknowledges these data and interrupts the CPU when 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 MCU and FPGA is depicted in Figure 3.

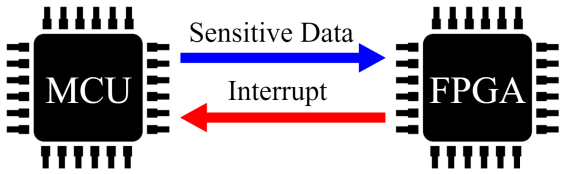


Fig. 3: The MCU-FPGA cooperation system for protection.

The program is instrumented so that single **`OUT/STR`** instructions 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).

For each sent data, the CFI monitor needs to receive the kind of the provided data as well, to perform the right operation on it. For this reason, the interface must have an *address bus* whose address will actually be interpreted as an *opcode* by the monitor.

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.

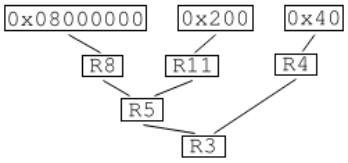
We refer as *target* of an edge to the BB pointed by that edge. From this definition, we can define *direct edges* and *indirect edges*. Direct edges are edges whose target is expressed as a label encoded within the instruction itself, while indirect edges are edges whose target is expressed by the value 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 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, if we assume that the entire code is already available in a single executable, and there will be no modules linked at runtime, then *the construction of that 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

```
MOV R8, #1
LSL R8, R8, #27
MOV R11, 0x200
MOV R4, 0x40
ADD R5, R8, R11
ADD R3, R4, R5
BX R3
```

(a)



(b)

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

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, with CFI solutions, which of these points is the right one according to the execution. In the case of an insecure backward edge, there is instead a way, because in addition to store all the possible destinations, it is also possible to store in the monitor the identifier of the next BB to be executed, so that the execution is then forced to return there.

Interrupt Service Routines

The assumptions made so far are valid only if one does not consider that actually the processor, in undefined moments of the execution, can jump to execute special routines to serve interrupt requests (Interrupt Service Routines, ISR). As explained in [45], there is no static analysis that can forecast in which order or where in the code these routines are 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 executable 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 unique 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.

At the end of the analysis and instrumentation process, the Assembly source of the firmware is recompiled, to produce the actual final version of the executable. At this point, 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* loads the instrumented version of the firmware and programs the FPGA, correctly setting the parallel interface in order to allow the runtime interaction. The online phase now starts,

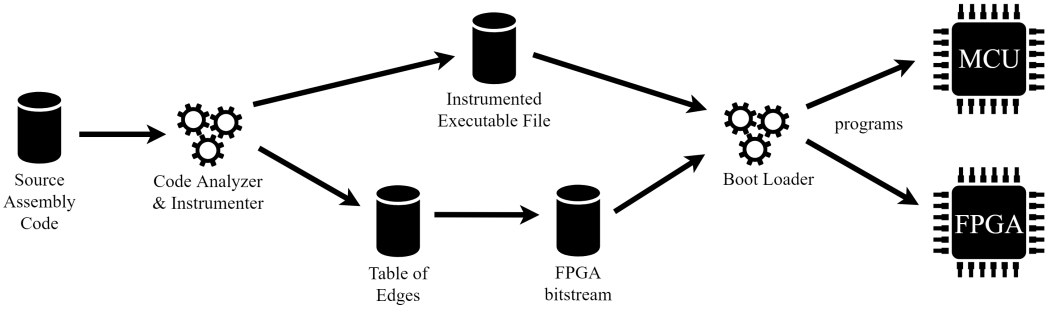


Fig. 5: The workflow of the analysis and instrumentation process.

with the FPGA acting as a monitor in response to control-flow information received by the instrumented program.

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 MCU to generate consequent reads and writes on these three storage blocks, as well as it verifies, through a set of comparators, that the data received are the expected ones. As an output, the unit controls the *interrupt line*, which notifies the MCU 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. With respect to this, 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.

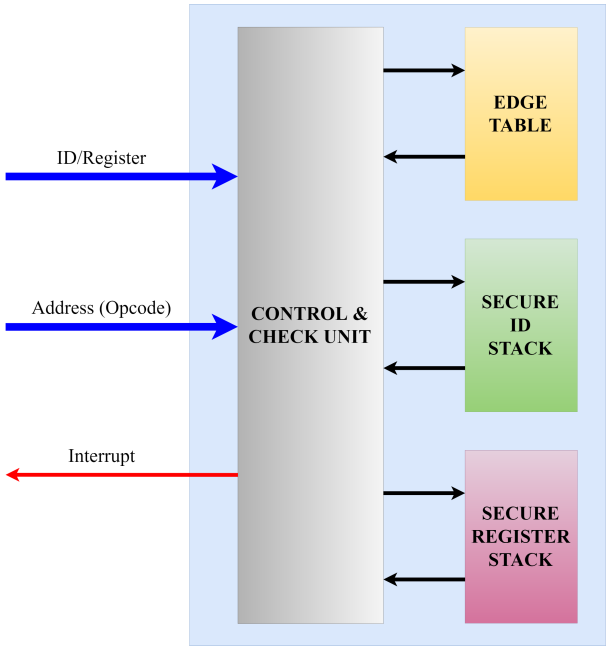


Fig. 6: CFI monitor block diagram.

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 and then protect it 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 warn the microcontroller as soon as possible of the attack. To achieve this, an intelligent encoding for the consented edges should be adopted, which allows for example 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 overhead in terms of 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 frequency at which MCUs and FPGAs operate should be properly tuned, so that the monitor is faster than an attacker.

VII. Experimental Results

In this Section, some preliminary experimental results obtained from example code analysis are presented[1].

[1][Note for the reviewer: We are currently gathering additional experimental result on these applications: https://www.x41-dsec.de/lab/advisories/x41-2018-003-pam_pkcs11/, https://www.x41-dsec.de/lab/advisories/x41-2018-004-libykneomgr/, https://www.x41-dsec.de/lab/advisories/x41-2018-002-OpenSC/. Experimental results will be available for the camera ready version of the paper, if accepted.]

The SEcube™ Open Security Platform

The proposed solution has been implemented and tested on the SE*cube*™ platform [5], an open security-oriented platform produced by Blu5 Group® The platform is implemented as a 3D SiP (System-in-Package) integrating three components:

- A STM32F4 microcontroller by STMicroelectronics™ ;
- A MachX02 FPGA by Lattice Semiconductor™ ;
- An EAL5+ certified Smart Card;

The device has been chosen because it offers important hardware security capabilities, such as a closed and protected physical chip which prevents the common physical attacks [32] [12].

Example Application

In order to test the proposed solution, a simple vulnerable application has been developed on the SE*cube*™ platform. The application consists in a program that responds to commands received through the UART serial interface. The intentionally-inserted vulnerability is presented in Figure 7.

```
void receive_command()
{
char in_cmd[4];
uint8_t in;
count = 0;
 while (in != '\n') {
  if (HAL_UART_Receive_IT(...) ... ) {
   in_cmd[count] = in;
   count++;
  }
 }
return;
}
```

Fig. 7: The vulnerable function present in our test application.

Proper checks on the input length are missing, so it is possible to overrun the `in_cmd[]` buffer corrupting the adjiacent content of the stack, included the return address. The result is that any address in code memory is reachable when the `return` is executed. After the code instrumentation and the implementation on the FPGA of the CFI monitor described above, it was no longer possible to redirect the execution stream. Even though the MachX02 device present on the chip is composed of 7000 4-bit Look-Up Tables (LUT), 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 table (1 entry per edge).

Code Instrumentation

Critical points identification (see Section VI) of both edges and context revealed the presence of:

- Backward insecure edges with single target;

- Forward secure edges to a routine ending with a backward insecure edge with multiple targets;
- Backward insecure edges with multiple targets.

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`** operation does not support an immediate address, so this must be first copied into a register.

In Table I, some data obtained by our test implementation are reported.

TABLE I: Experimental Preliminary Results

Assembly instructions of non-instrumented code	4123
Control-flow transfer instructions	525
Backward insecure edges with single target	19
Backward insecure edges with multiple targets	38
Protected edges	57
Instrumented ISR	2
Instructions added for edge protection	587
Instructions added for context protection	156
Assembly instructions of instrumented code	4866
Percentage of code overhead	15.2%
Total number of FPGA LUTs	6864
Occupied LUTs	185 (3% of total)
Total amount of FPGA Memory	240Kb
Occupied FPGA Memory	165Kb (69% of total)

VIII. 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 (*OS-based*, *purely software-based*, *hardware-based*). Using a mixture of binary instrumentation and hardware-based supervision, the solution entrusts the binary enforcement with the sole task of informing the monitor about the status of the CFG through simple additional **`write`** instructions at critical points, and the hardware monitor with the conservation of the information about the CFG and the part of computation for the validation, thus obtaining advantages in terms of both isolation and performance. In addition, the monitor is implemented on a reconfigurable hardware device connected externally via a standard parallel interface. This frees the solution from constraints on specific components and from the need to modify the internal structure of the microcontroller to support the defense. The only constraint is the presence of a parallel configurable interface to the outside, but it is highly unlikely to find microcontrollers not supporting it.

Furthermore, our solution is applicable to any microcontroller system in principle, because the assumptions made in Section VI never descend into the particular of a specific architecture, except for giving examples. *Mutatis mutandis*, this technique can be ported to any platform. Moreover, far more important, the technique can also be applied to all those systems that are already in the field (*legacy*). It is only required to connect a FPGA to the microcontroller, patch the firmware with the instrumentation, and enclose the system in a physically-protected environment.

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] 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.

[7] 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.

[8] Arm. TrustZone - Arm Developer. https://developer.arm.com/ip-products/security-ip/trustzone. [Online; accessed 13-November-2019].

[9] S. Bhatkar, D. DuVarney C, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, volume 12, pages 291–301, 2003.

[10] 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.

[11] 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.

[12] 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.

[13] C. Bresch, D. Hély, A. Papadimitriou, A. Michelet-Gignoux, L. Amato, and T. Meyer. Stack redundancy to thwart return oriented programming in embedded systems. *IEEE Embedded Systems Letters*, 10(3):87–90, Sep. 2018.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In A. Prakash and I. Sen Gupta, editors, *Information Systems Security*, pages 163–177, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[21] 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.

[22] 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.

[23] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attack. 2014.

[24] 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.

[25] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, , and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. 98:5–5, 01 1998.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro. Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 15–27. ACM, 2019.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] Z. Guo, R. Bhakta, and I. G. Harris. Control-flow checking for intrusion detection via a real-time debug interface. In *2014 International Conference on Smart Computing Workshops*, pages 87–92, Nov 2014.

[38] 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.

[39] 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.

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

[41] K. S. Kumar and D. Malathi. A novel method to find time complexity of an algorithm by using control flow graph. In *2017 International Conference on Technical Advancements in Computers and Communications (ICTACC)*, pages 66–68, April 2017.

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

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

[44] 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.

[45] 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.

[46] T. Nyman, J.E. Ekberg, L. Davi, and N. Asokan. Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 259–284. Springer, 2017.

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

[48] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, 2013.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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].

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

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] Intel Software Developer Zone. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf. [Online; accessed 13-November-2019].