Recommendation for a holistic secure embedded ISA extension

Florian Stolz[1], Marc Fyrbiak[2,3], Pascal Sasdrich[1], and Tim Güneysu[1]

[1] Ruhr-Universität Bochum, Bochum, Germany
`{firstname.lastname}@rub.de`
[2] Max Planck Institute for Security and Privacy, Bochum, Germany
`marc.fyrbiak@mpi-sp.org`
[3] emproof GmbH, Bochum, Germany

Abstract. Embedded systems are a cornerstone of the ongoing digitization of our society, ranging from expanding markets around IoT and smart-X devices over to sensors in autonomous driving, medical equipment or critical infrastructures. Since a vast amount of embedded systems are safety-critical (e.g., due to their operation site), security is a necessity for their operation. However, unlike mobile, desktop, and server systems, where adversaries typically only act have remote access, embedded systems typically face attackers with physical access. Thus embedded system require an additional set of defense techniques, preferably leveraging hardware acceleration to minimize the impact on their stringent operation constraints. Over the last decade numerous defenses have been explored, however, they have often been analyzed in isolation. In this work, we first systematically analyze the state of the art in defenses for both software exploitation and fault attacks on embedded systems. We then carefully design a holistic instruction set extension to augment the RISC-V instruction set architecture with instructions to deter against the threats analyzed in this work. Moreover we implement our design using the *gem5* simulator system and a binary translation approach to arm software with our instruction set extension. Finally, we evaluate performance overhead on the *MiBench2* benchmark suite. Our evaluation demonstrates a ROM overhead increase of 20% to defeat the aforementioned attacks.

Keywords: Embedded Security · Physical Attacks · ISA Extension

1 Introduction

With the ubiquitously expanding Internet of Things (IoT), the demand for embedded devices continuously increases. However, such an ubiquitous presence of embedded systems in also security-relevant appliances inevitably increases the potential for attacks. Through physical access, adversaries can particularly attack digital devices and security-critical systems through implementation attacks, such as Side-Channel Analysis (SCA), Fault Injection Attack (FIA). Along with (limited) software-induced attacks, the range of threats that modern embedded devices face is broad and multifaceted. Physical access allows adversaries to perform glitching attacks, which may lead to bit flips in the fetched instructions. These faulty bits may cause erroneous instructions to be executed or may even change the semantics to a No-Operation (NOP), effectively skipping an instruction [2]. Especially in the context of cryptography, such glitches can have severe consequences such as key leakage via Differential Fault Attack (DFA) [24]. Furthermore, glitch-induced NOPs by physical adversary may violate the control-flow of the program, which in term can leak secrets. On the software side, adversaries can mount software-based attacks to manipulate the control flow of the program, for example, by overwriting the return address. Over the last decades, many countermeasures [3, 11, 15, 22, 33, 34] to these threats have been developed, but have mostly been studied in isolation. However, as seen above, in the case of embedded systems, the adversary can mount an attack using many different techniques. Therefore, an embedded system has to employ a combination of defenses to deter these adversaries. Simply stacking different approaches on top of each other may give rise to inefficiencies, as they may reimplement similar primitives instead of sharing a common base. Instead it is preferable to determine an efficient set of instructions which can achieve a maximum of security by reusing primitives.

Goal and Contributions. In this work, we focus on a embedded system defenses to protect against both software-based exploitation and glitching attacks simultaneously. Our goal is to design an instruction set extension with a particular focus on RISC-V to minimize performance impacts. To this end, we first systematically analyze the state of the art for both aforementioned attack strategies. Based on elaborated insights, we then carefully design our instruction set extension to leverage synergies between different defenses, i.e. employing a glitching defense to facilitate higher-level defenses. By hashing the current instruction stream, we create a label-based Control Flow Integrity (CFI) scheme to protect forward-edges as well as a pointer protection scheme to defend backward-edges against Return-Oriented Programming (ROP) attacks. Finally, we implement and evaluate our instruction set extension using binary translation and demonstrate an average memory overhead of 20% and average performance overhead of 28%. In summary our contributions are:

– **Systematic Analysis.** We carefully analyze state-of-the-art hardware-accelerated defenses against instruction glitching attacks and memory corruption vulnerabilities. Based on our analysis, we then work out defense combinations to leverage their advantages to maximize security guarantees, while minimizing potential performance impact.
– **Novel Hardware Extension.** Based on our systematic analysis, we design a novel instruction set extension for RISC-V that defeats the aforementioned attacks and induces minimal overhead. In particular, our extension leverages an anti-glitching defense that ensures basic block instruction stream integrity. Each basic block hash is then used to implement a label-based CFI defense to protect forward control-flow edges. Furthermore, we use a pointer protection to secure backward control-flow edges.

– **Evaluation.** We evaluate our recommendation using the *gem5* simulator and the *MiBench2* benchmark suite. We then compare our solution to existing works, which aim to secure embedded systems against similar threats, and find that our solution has a 39% lower memory overhead while having a 81% higher performance overhead.

2 Technical Background

In the following, we provide a concise background on most prominent attack vectors for embedded systems, namely (1) software-based attacks via code injection and re-use, and (2) fault attacks via glitching.

2.1 Code Injection & Reuse Attacks

Even though processing technology made significant progress, embedded systems are still typically constrained in both resources and features. Moreover, C and C++ are still the predominant languages and they do not provide any memory safety features and thus improper use of memory allocations leads to catastrophic exploits (e.g., buffer overflows on both the stack and heap can be used to mount code injection attacks). Advances in the last 20 years, such as DEP and Stack Canaries, made this type of attack more challenging to perform. Nowadays, these countermeasures can also be found in embedded CPUs, which usually offer basic memory protection in form of non-executable memory regions. This restricts attackers to leverage so-called code reuse attacks, f.i., ROP-based exploitation [9].

2.2 Glitching Attacks

Adversaries are especially powerful when granted physical access to the target device as this enables to challenges various assumptions (e.g., the integrity of the instruction stream). By performing a *fault attack*, for example, via *glitching* the clock source or via electromagnetic pulse, an adversary can disturb instruction stream integrity. Bitflips caused by glitches manipulate the data embedded into instructions or change instruction semantics entirely. For example, under certain conditions an instruction can be changed to an NOP instruction. Note that this has severe impacts on the control flow of the program if the skipped instruction is a branch or a comparison [26]. In cryptographic algorithms, such glitches can induce exploitable weaknesses to break all security guarantees. For example, flipping bits during the key addition step may allow attackers to perform a DFA. The possibility of glitching attacks and their effects on program execution have been studied extensively times in literature. Most recently, Spensky *et al.* [26] analyzed physical attacks under simulated and real conditions. For already existing hardware architectures, special programming techniques are use to prevent such vulnerabilities. Defensive programming techniques were previously analyzed by Wittemann *et al.* [30]. Furthermore, compiler modifications such as presented by Barry *et al.* [3] deter glitching, for example, via instruction duplication. Spensky *et al.* [26] combined different methods to defend against glitches as a *LLVM* extension, including the previously mentioned techniques of code redundancy. A major problem of these approaches is the induced overhead both in the code size and time domain.

3 Preliminaries

We now introduce the fundamentals for the scope of our work, including the assumed system model and adversary model. Based on the models, we then define the security goals that should be achieved by a holistic Instruction Set Architecture (ISA) extension in the context of embedded systems.

3.1 System Model

As outlined before, we focus on a common embedded system model (e.g., typical for many IoT applications): a resource-constrained System-on-a-Chip (SoC) including a processor and other important peripherals such as Random-access memory (RAM) and Read-only memory (ROM), f.i. with memory in the sub one megabyte range and processing speeds of up to 200 MHz. Moreover we assume a system with an Memory Protection Units (MPUs) to offer basic memory protection. Examples of such systems are the Cortex-M range by *ARM* and many RISC-V based products. Throughout this work, we assume that the software runs in a *bare-metal* fashion on the embedded systems, i.e. no real-time operating system is available for the sake of simplicity. However, we want to emphasize that the with a context switch, the real-time operating system support can be added as well.

3.2 Adversary Model

We assume an attacker with physical access to the target system, i.e. to analyze the Printed Circuit Board (PCB) and peripherals. Moreover, we assume that the attacker is able to mount physical attacks by means of fault injection via glitching. However, attacks on the microarchitecture itself and fault injection via laser are out-of-scope of our work. Consequently, we assume the integrity of the integrated RAM/ROM. Exploitation is thus only possible by manipulating signals or sending malicious inputs to the device, i.e. to leverage a software bug for exploitation. The high-level goal of the adversary is to exploit a given device because of private or economic incentive (e.g., forcefully unlock (unintended) features).

3.3 Security Goals

The nature of our adversary model implies protection against various attacks vectors that can be leveraged for exploitation.

Anti-Glitching. Firstly, the integrity of executing instructions should be guaranteed. Otherwise, an adversary can introduce disturbances via glitching to affect the execution. Note that in the context of embedded systems, anti-glitching techniques should be tightly coupled to other defenses, such as CFI, as glitches itself are able to lead to an invalid control flow.

Control Flow Integrity. Secondly, (arbitrary) attacker-controlled code execution should be prevented. Note that a properly configured Physical Memory Protection (PMP) on RISC-V processors can already stop basic code injection attacks by disallowing execution in certain regions, such as the stack. However, code-reuse attacks, such as ROP, are still possible. Thus, CFI is a vital requirement for system security. Especially on resource constrained devices, hardware-accelerated CFI on the ISA level may be viable, as code instrumentation creates significant performance impact.

Memory Safety. Lastly, memory corruption should be prevented. Memory unsafe languages such as *C* facilitate exploitation of software bugs (e.g., to overflow memory buffers or write to unwanted memory locations). Note this leads to control-flow changes that may not be detectable by a CFI scheme. Under our adversary model, memory safety has to be combined with other defenses, since perfect memory safety does not eliminate CFI violations as instruction-skips may also be used to mount attacks.

4 Literature Study

We now provide a concise summary of the state of the art of various defenses that have been proposed over the years. Based on our summary, we then discuss which defenses achieve best synergy effects in our system and adversary model.

4.1 Glitching Defenses

The increasing reliance on embedded systems in, for example, IoT appliances or critical infrastructure also prompts for increased security measures in these devices. Considering the potentially severe consequences of glitching attacks as mentioned in Section 2.2, several hardware accelerated countermeasures have been developed. In the context of embedded systems, defenses against fault attacks are often combined with other techniques to facilitate CFI. This stems from the fact that skipping control flow instructions can cause control flow violations. In this section, we exclude CFI and focus on instruction stream integrity. Note that we discuss how these primitives may facilitate CFI in detail in Section 4.2.

In general, we divide glitching attack defenses into three groups based on their underlying mechanism: anomaly detection, instruction chaining and instruction hashing. As noted before, we focus especially on instruction skips caused by glitching.

Anomaly detection. This defense techniques aims to alert the processor about a possible glitch attack by observing its environment and scanning it for potential indicators of an attack, such as voltage drops. Yuce *et al.* [33] demonstrated *FAME*, a co-processor which triggers a software handler upon detecting a fault. The co-processor is composed of a fault detection unit, which monitors several aspects of the Central Processing Unit (CPU), such as the clock signal. After detecting a fault, *FAME* stops the execution of the main processor. It enters a *safe mode*, where a trap handler can recover the fault. A major advantage of this approach is its low footprint in code size and execution time. The original program does not have to be recompiled and only a fault handler has to be added.

Instruction chaining. Bitflips manifest themselves in a changed instruction word, for example, changing a branch to a NOP. These changes usually only have a limited impact on the surrounding instructions, as each instruction is essentially executed in isolation. Therefore, making instructions depend on each other can spread the effects of glitching attacks out, which usually leads to unrecoverable corruption of the instruction stream and thus deter a possible attack. This approach is used by Werner *et al.* [27] to protect processors from physical attacks. They describe a mechanism based on authenticated encryption, which accumulates a state based on the execution history and is used to decrypt each instruction as it enters the pipeline. A fault will be detected when an invalid instruction is decoded. Alternatively, an integrity verification may be performed to stop randomly decrypted instructions from executing. Similarly the works by Savry *et al.* [22] and de Clercq *et al.* [8] use a mask to decrypt instructions. The first approach uses a static mask, which changes based on a permutation during the execution, the latter work uses a mask based on several values such as the previous program counter location. Notably, the masks employed by Savry *et al.* are not depending on the execution history, which may allow for malicious modifications using dedicated managing instructions. However, the integrity of the initial mask is secured via a Message Authentication Code (MAC).

Instruction hashing. Hashes are commonly used to check the integrity of data. Thus, hashing instructions is a straightforward method to monitor the integrity of the instruction stream. Fei *et al.* [13] employ a standalone hardware monitor, which accumulates a hash during the execution of a single basic block. At the end of each block, the hash is compared to an internal table storing the expected hash. In case of a mismatch, an exception is raised. The authors mention the use of *MD5* or *SHA-1*, however, their demonstration uses a simple *XOR* hash. Rodriguez *et al.* [21] developed an architecture, which stores the expected hash and further attributes in the instruction stream. Similarly to the previous solution, the computed hash is compared to the expected hash. The block length is encoded into the attributes to prevent attackers from skipping the hash comparison. The authors suggest a *XOR* or Linear-Feedback Shift Register (LFSR) as the hash function. A similar approach was used by Ohlsson *et al.* [19], who use a Cyclic Redundancy Check (CRC) as the hashing function. Werner *et al.* [28] systematically evaluated the requirements for an appropriate hash function and found that a CRC is best suited for instruction hashing. A major problem of these approaches compared to the other fault defenses, is their latency. As the hash is only checked at the end of a block, the fault goes unnoticed for the remainder of the basic block. This can be accounted for by embedding a small hash into each instruction, which is checked during the execution of each instruction, as proposed by Wilken *et al.* [29]. However, this approach either requires a complete ISA redesign or some kind of hardware unit with access to each hash. The latter was implemented by Werner *et al.* [28] as a memory-mapped peripheral on a *ARM* processor.

Discussion Compared to software-only solutions, ISA and hardware-level can offer considerable overhead reductions. Nevertheless, not all of the previously mentioned techniques may be combined intuitively with other solutions or come with other downsides. Standalone hardware monitors such as *FAME* [33] or the proposal by Fei *et al.* [13] have the advantage that they do not require modification to the ISA itself. However, these monitors have to constantly detect basic blocks, for example, by scanning for branch instructions. It would thus be preferable to give the processor built-in capabilities to differentiate between basic blocks. Furthermore, internal storage required by these solutions is limited and needs managing and data swapping, as described by Fei *et al.*. In contrast to this, integrated instruction chaining or hashing avoid these problems and come with the advantage of being building blocks for other structures. Many of the previously proposed schemes are used to implement protection mechanisms such as CFI. They are thus ideal candidates for a holistic security extension. The main differences between chaining and hashing lie in their latency. Compared to hashing, chaining inhibits low latency as a bitflip will immediately result in randomly decrypted instructions. Yet this behavior may not always be desirable, especially if the processor may be in an elevated execution state. Therefore, further modifications are required to prevent random instruction execution. Furthermore, chaining is often based on encryption, which by itself usually results in major performance degradation or hardware overhead. Instruction hashing avoids these specific problems by leaving the individual instructions intact and performing the hashing operating in parallel. Additionally, the hashing function may be more compact in hardware than an encryption scheme. The latency problems of instruction hashing may be solved by using continuous hashing at the cost of a major ISA redesign.

4.2 CFI

Not only physical attacks pose a serious threat to embedded devices, but also more conventional software-based attacks. Whereas straight-forward code injection attacks are stopped by using technologies such as the MPU, more complex attacks like Code Reuse Attack (CRA) require additional defenses. CFI has been thoroughly explored as a countermeasure against such CRAs. This technique aims to enforce the Control Flow Graph (CFG) of a program by checking the forward edges, created by jumps, and the backward edges, caused by returns. A *fine-grained* approach strictly enforces the CFG, but usually creates a large performance overhead, whereas *coarse-grained* CFI relaxes the rules to gain performance at the cost of security [9].

We divide existing hardware accelerated solutions into three categories: State-based, Policy-based and Heuristics-based. However, in practice many solutions combine techniques from two or all categories.

Policy-based. During its normal execution, a program usually follows some predictable patterns. For example, by definition, a branch targets the entry of a basic block. This rule is broken by CRAs, which execute small code snippets within basic blocks. Thus, enforcing this rule thwarts some control-flow attacks and is used in many solutions like Intel CET [25], which uses the `ENDBRANCH` instruction to terminate an indirect branch. Alternative solutions prohibit arbitrary branches between functions, for example, via a label, which is placed at each call/return site and is checked during each control-flow transfer. If the expected label does not match, a control-flow violation occurs. This approach is used by Christoulakis *et al.* [7] who employ the branch-delay slot to efficiently load the expected label during a branch. The label can also be generated implicitly using the *Instruction hashing* explained in Section 4.1 based on the execution history.

State-based. If an attack follows the rules outlined above, it will stay undetected degrading the security of the system. This may happen if two functions have the same label, which allows for two legit backward edges. To overcome this problem many CFI solutions include a state, which is checked on a regular basis. The most popular solution is a Shadow Call Stack (SCS) [6]. Every call pushes the return address to the regular stack as well as to the SCS. Upon a return, the addresses on the stack and SCS are compared. If they do not match, an exception is thrown. The SCS may be implemented as a hardware stack or as a second stack located in RAM. Davi *et al.* [11] introduced an alternative approach by forcing the function to return to an active call site. Each function is assigned a label, which is activated by an instruction at the start of the function. When returning from a callee it is checked if the label is still active. If not, an adversary changed the return address to a function which was not previously active. Alternatively, the *instruction chaining* approaches from Section 4.1 can be used to implement CFI. Encrypting instructions and making them dependent on the previous instructions stops the attacker from arbitrary executing existing code.

Heuristics-based. Lastly, hardware monitors can be employed, which screen the execution history for unusual behaviour, such as short instruction sequences followed by returns indicating a ROP attack. A major problem however is the correct selection of the underlying heuristic. Solutions such as by Kayaalp *et al.* [15] use multiple thresholds to allow for variable gadget length, which lowers the false positive rate.

Discussion A good CFI solution should provide some form of forward as well as backward-edge protection. Heuristic-based solutions cannot provide such a protection as they detect attacks based on the execution characteristics. Furthermore, it has been shown that monitors which aim to detect short instruction sequences between branches, which are typical for short gadgets, can be circumvented. Thus, heuristics do not seem to be a viable candidate for general-purpose microcontrollers. Most CFI solutions use a mixture of different techniques to achieve both backward- and forward-edge protection. The most promising candidates for backward-edge protection are SCSs as well as HAFIX [11]. However, both require special attention to support common programming paradigms such as recursion and, in the case of SCSs, exceptions. Ideally, SCS data should be placed in processor internal memory, which however is limited, in term requiring some kind of on-demand loading. Therefore, to minimize overhead, a label-based policy seems viable for embedded systems. This also allows us to reuse *instruction hashing*. Notably, this only protects forward edges. Backward-edge violations occur when overwriting the return address, thus requiring a memory protection as discussed below.

4.3 Memory Integrity

In case of a remote attacker in a IoT scenario, a memory vulnerability, such as a buffer overflow, may be used to overwrite data like return addresses on which the control-flow depends. In this paper we refer defenses which guard against such memory attacks as *memory integrity*. As before, we divide existing work into three categories: detection, pointer protection and tagging.

Detection. Buffer overflows actively change the data surrounding the original buffer. Many solutions aim to detect such overflows by placing control values in front of important data like the return address. These are commonly referred to as *stack canaries*. Before each return, the control value is compared to its expected value. If the values do not match an exception is raised [10].
De *et al.* [12] developed a RISC-V extension, which employs a Physically Unclonable Function (PUF) and a binary secret to generate canaries which are placed at each buffer. Thus, their detection method can detect buffer overflows even if they do not overwrite the return address, which is usually the only the checked location.

Pointer Protection. Alternatively, pointers may be protected so that they cannot be simply overwritten by the attacker. In 2016 ARM introduced *Pointer Authentication* as an ISA extension to ARMv8.3-A [23]. It is based on the realization that the upper bytes of a 64-bit pointer are essentially unused and can store additional information. They place a short MAC, also referred to as Pointer Authentication Code (PAC) into the upper bits of the pointer, which is computed using the current context, such as the stack pointer, and domain specific keys. Before using the pointer, an authentication instruction verifies the PAC and makes the pointer invalid if the process fails. On the contrary, *Pointer Encryption* modifies the whole pointer by encrypting it before placing it into memory. The approach by Zhu *et al.* [34], uses a *GCC* extension which automatically protects relevant pointers by applying a XOR-encryption with a key derived from the memory location and a dynamic runtime key. A downside of this approach is the random execution which is inevitably caused by any kind of corruption or manipulation of the encrypted pointer, as the plaintext pointer will be random. Lastly, *Pointer Capabilities* add specific access rights and constraints to each pointer, so that out of bounds writes become impossible. The CHERI ISA [32] represents a major project which uses a 256-bit pointer format to encode various information about each pointer, such as the base address, length and permissions.

Isolation. In the following, we briefly describe approaches which isolate memory regions from each other, so that they cannot interfere. Most microcontrollers already offer basic functionality for memory isolation through a MPU on ARM or the PMP on RISC-V. However, the amount of memory regions as well as available access (e.g., user-mode and supervisor-mode) are limited. Furthermore, even in the presence of unique process identifiers, process internal memory isolation remains a problem. The ARM MTE extensions [1] introduced in ARMv8 provide memory tagging by assigning a 4-bit tag to every 16 bytes in memory. Additionally, each 64-bit pointer stores a corresponding tag in its most significant byte. Only if the tags match, a read or write are possible. This can not only protect against buffer overflows, but also against use-after-free attacks. The approach by Bradbury *et al.* [5], implements tagging for RISC-V and uses 2-bit tags for every 8 bytes in memory. Unlike ARM's solution, the tags do not correspond to a key, but to access permissions such as read- or write-only. Kim *et al.* [16] presented `RIMI`, which extends upon RISC-V's PMP and add instructions which only allow control-transfers and load/stores inside one of two domains. All relevant instructions are duplicated for *domain0* and *domain1*. This approach is especially viable on embedded systems with limited resources, as no additional tag bits are required. Instead, the isolation is achieved through specialized instructions as well as existing and added hardware units.

Discussion Many memory protection schemes are built with larger systems in mind, which becomes a problem in the context of resource-constrained embedded systems. Capability-based systems can offer a fine-grained protection at the cost of increasing the necessary space required for a single pointer and increased complexity. Even compressed capability formats such as CHERI Concentrate [31] take up 64 bits for 32-bit architectures. A similar problem applies to ARM PAC as well as ARM MTE as they abuse the fact, that the upper bytes of 64-bit pointers are typically unused. On embedded 32-bit platforms placing a sufficiently secure MACs becomes impossible. Most of the time, the whole 32-bit space is used with only a few bits being available. Isolation through mechanisms such as `RIMI` are suitable for embedded systems, but require a general rewrite of the software. Memory Tagging in general seems to be a good fit for embedded systems but comes with some downsides such as limited tag space. Naturally, a trade-off must be made between available tag bits, which allows for more fine-grained control, and the acceptable memory overhead. For example, ARM's solution using 4 bits for every 16 bytes necessarily occupies 8 kB out of a 256 kB RAM module. Furthermore, a software or hardware component is required to manage the tags and schedule them accordingly. Therefore, to keep the original pointer size on 32-bit systems only pointer encryption and overflow detection are viable. Both solutions can be made context dependent, which is easily possible by using, for example, the current CFI state creating a strong synergy between previously discussed defense components. However, as the attacker can deduce the current CFI state, it necessary to introduce some kind of processor secret, so that the attacker cannot guess the canary or decrypt the pointer. An advantage of pointer encryption is that no memory overhead is created, as long as the cipher has the same block size as the pointer.

5 Our Recommendation

Based on the results of our literature study we now present our recommendation for a holistic ISA extension. We explain our approach by dividing programs into their individual levels mainly the basic block level, the functional level and the global level. For each level we name the main threat and show how it can be countered using our extension. Using a bottom-up approach, we show how we can reuse primitives from lower levels to facilitate other higher-level protection mechanisms.

5.1 Basic Block Level

We start by analyzing a single basic block, which on its own, consists only of sequence of instructions. On this level, an adversary may be interested in either changing the semantics of one or multiple instructions or skipping them entirely using glitches. In Section 4.1 we showed several defenses against glitch attacks.

In accordance with our discussion in Section 4.1, we propose the usage of *instruction hashing* as a countermeasure on this level. Compared to the other proposed solutions, hashing allows us to keep the original instructions unchanged and only add the operations necessary for the hashing process. Furthermore, the hash does not only give us information about the integrity of a basic block, but also a value which characterizes the execution history, albeit in a limited form. This can later be used for different protection mechanisms. Thus, several additions have to be made to the processor: Firstly, we have to provide a hardware unit that is closely connected to the processor pipeline which is responsible for the hashing. Secondly, each basic block has to augmented with a `CHECK` instruction, which embeds an expected hash which is compared to the actual computed hash. If they do not match, an error will be raised. The `CHECK` instruction can theoretically be placed arbitrarily, however, in this context only two positions make sense. Either the hash is checked at the end of a basic block or at the start. A position between the start and the end of a basic block would not cover the whole instruction sequence. We placed the CHECK instruction at the beginning of each block as shown in Figure 1, so that it can be later used as a CFI mechanism. Therefore, in practice the integrity of basic block BB_t is checked in the following block BB_{t+1}. A straightforward approach for the hashing itself would be a simple XOR function, however, as pointed out by Werner *et al.* [28], a CRC function can provide better security by increasing the complexity for the attacker to mask his glitch attempt.

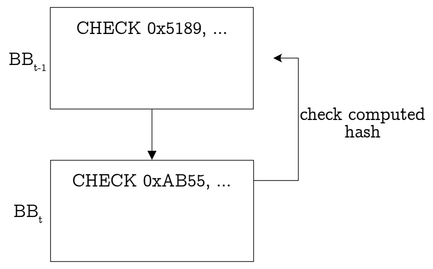

Fig. 1. The CHECK instruction provides a simple interface to determine the correctness of the previous basic block.

5.2 Function Level

A function is comprised of several basic blocks, however the execution path between them may not be linear but consist of more complex paths created, which may cause some blocks not to be executed. Here, an attacker can try to manipulate the intra-functional control flow, for example, by skipping branch instructions and causing a fall-through.

The CHECK instruction introduced in the previous section already implicitly provides a protection mechanism against intra-functional control flow violations. However, typical functions do usually not consist of a pure linear execution path but incorporate branches and loops. In this case, a simple scheme consisting of CHECK instructions is not sufficient, because the execution history differs for each side of a branch resulting in two different hashes. Thus, once the execution rejoins the common path of the function, the CHECK instruction will fail. Consequently, we need to be able to synchronize the hashes. Therefore, if we identify a basic block with multiple predecessors, we select the hash of one predecessor as the expected hash and add a CORRECT instruction to each other predecessor, which synchronizes the hash to match the expected one. Similarly to other proposal, we may use the XOR function to correct the hash. An overview is given in Figure 2.

The above approach exhibits several weaknesses, which have to be addressed to make it suitable as an effective defense method. First, in case of a branch both successors of the basic block will necessarily have the same hash. Therefore, it may be possible to glitch the processor state and execute, for example, the

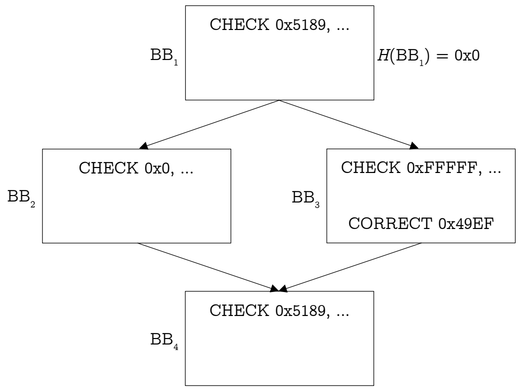

Fig. 2. Example of a `CORRECT` instruction being issued to synchronize the hashes. Note that BB_2 and BB_3 have inverted hashes to prevent a simple branch target change.

not taken path instead the *taken* path. However, it is possible to diversify the hash by taking the *taken / not taken* decision into account when generating the hash. We want to note, that this may make hash synchronization more complicated in case of a complex control flow. This problem may be resolved by the compiler by inserting small trampolines including additional `CORRECT` instructions to properly synchronize the hash. Second, if the attacker can cause the execution of `CORRECT` instructions, he can change the hash state arbitrarily. In consequence, the placement of these instructions should be limited. The processor can be augmented with a Finite State Machine (FSM), which only allows corrections when in a specific state. From the previous description it is clear, that a correction should only be issued before a basic block ends. A basic block either ends in a fall-through, in which case the following instruction is a `CHECK`, or in a branch, which then it term is also followed by a `CHECK`. Enforcing these rules via a FSM restricts the attacker from freely executing corrections. However, there are several other attack vectors which can be closed by enforcing additional rules. As an example, an adversary can attempt to skip the branch at the end of a basic block and the following `CHECK` instruction, thus creating a linear control flow. This can be counteracted by making the processor aware of the basic block length and throwing an exception if it does not encounter a `CHECK` after the specified length similarly to ISIS [20]. Here, a trade-off has to be made. Encoding the length into the `CHECK` instruction leads to a shorter hash length or scarifies other information bits, which potentially makes it easier to cause hash collisions. Instead, we propose to add an internal counter to the processor, which counts the instructions of each basic block and specifies a maximum amount. If the basic block does not reach its end before hitting the specified threshold, we assume that a manipulation took place and throw an error. This has the advantage, that we do not sacrifice encoding space and furthermore, the counter may be accessible to the programmer via fuse-bits to allow for device-specific modifications.

5.3 Global level

On a global level a program consists of multiple functions. Here, well-known software-based attacks may take place, such as CRAs or code-injection attacks. Additionally, control-flow bending may be used to divert the execution along an alternative but valid path along the CFG. Thus, CFI as well as memory integrity as described in Section 4.2 and Section 4.3 are of most importance on this level.

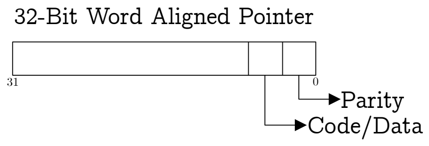

Fig. 3. Structure of a protected pointer exploiting the aligned property to store additional information.

The included `CHECK` and `CORRECT` instructions already provide an CFI protection mechanism. Each hash can be treated as the label of a specific basic block or function. The coarseness of this CFI protection is dependent on the hash length. For example, an 8-bit hash only allows for 256 values. It is safe to assume, that complex embedded software consists of more than 256 basic blocks, which will inevitably lead to some form of CFG relaxation. This means, that multiple blocks will be assigned the same label. Therefore, the control flow can be diverted to any basic block which shares the label of the original destination. Even with larger hash sizes, this can become problematic. According to our previously explained mechanism, a synchronization has to take place at the source locations and/or at the end of the target function. Therefore, all source locations create the same target label using corrections and the function creates one label which is shared between all return destinations. To protect against this threat we propose the introduction of a *pointer encryption* scheme to hinder the attacker from injection valid return addresses. *Pointer Authentication* akin to ARM's PACs are only feasible on systems with unused address bits. On embedded systems, the predominate architecture is however 32-bits of which almost all bits are required. However, we can combine elements of both approaches to facilitate a mechanism which can be used in the same way as ARM's PAC. Whenever a function is entered, we encrypt the return address in the link register using a key comprised of the expected hash at the usage location and an internal secret, which may be a random number generated during the boot process of the microcontroller. Then, at the intended usage location, e.g. the end of the function, we decrypt the return address using the hash of the current location and the internal secret. Only if the hashes match, the correct decryption key is generated. Therefore, the attacker has no direct control over pointer. Under the assumption of an architecture which only allows word-aligned accesses, this mechanism can be enhanced even further. In this case, the two least significant bits are always zero. Therefore, we can encode further information into the pointer as shown in Figure 3. One bit can be used to indicate whether the pointer refers to code or data and the other bit can be used to perform a simple parity check. This allows programs to differentiate between pointer types and create secured data and code pointers which are only usable at their intended locations similar to ARM PAC. The protection mechanism can be seen in Figure 4.

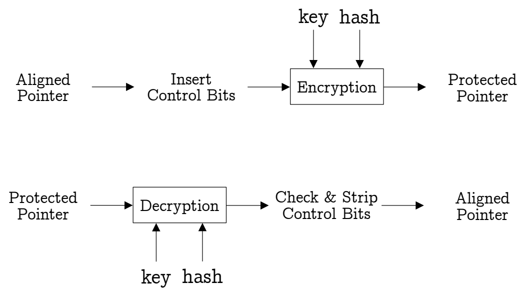

Fig. 4. Overview of the pointer protection process.

Furthermore, we can enforce some stricter CFI rules using our FSM and additional information in each `CHECK` instruction as seen in Figure 5. We can use one bit each to encode whether the basic block is an function entry, which prevents targeting arbitrary basic blocks using a call, and we can mark all basic blocks which are a return target. Thus, disallowing arbitrary returns to basic blocks which are not actual return locations.

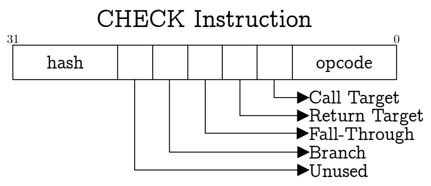

Fig. 5. Internals of the `CHECK` instruction showing the hash as well as bitfields specifying basic block specifics.

6 Proof-of-concept

We now demonstrate the practicability of our approach by implementing the proposed instruction set extension design for *RISC-V* and evaluate it using the cycle-accurate *gem5* simulator [17].

Instruction	Operands	Description
`CHECK`	`20-bit hash, 5-bit properties`	Checks hash and defines properties of current block
`CORRECT`	`20-bit value`	Performs a correction to synchronize hash
`ENCCPTR`	`20-bit hash, register`	Generates a code pointer out of aligned pointer and encrypts it
`ENCDPTR`	`20-bit hash, register`	Generates a data pointer out of aligned pointer and encrypts it
`DECCPTR`	`register`	Decrypts pointer and strips code pointer data
`DECDPTR`	`register`	Decrypts pointer and strips data pointer data

Table 1. Instructions required to implement our recommendation.

6.1 ISA Implementation and Simulation

We chose the RISC-V 32-bit standard as our target ISA because of its built-in support for user-defined instructions and its rising popularity in embedded systems. In total, we require six new instructions: two instructions to implement anti-glitching and CFI as well as four instructions to handle the pointer protection. Furthermore two new registers are added which hold the current hash state and the processor secret, see Table 1 for an overview of the additional instructions. The hash length was set to 20 bits, so that the `CHECK` instruction is able to hold the hash as well as state information for each basic block. We added support for these custom instructions and registers to the *gem5* simulator. Note that at the time of writing, *gem5* only supports the 64-bit RISC-V standard, however, all 32-bit opcodes stay valid, thus allowing us to simulate 32-bit code. Furthermore, we also added the required FSM and encryption/decryption module to the simulated processor to enforce the rules outlined in Section 5.2 as well as Section 5.3 and to enable support for the pointer encryption. As we operate on 32-bit pointers, we chose the hardware-optimized *SIMON* cipher [4] with a block length of 32 bits and a key length of 64 bits. The encryption uses a connotation of the expected hash with a 44-bit processor secret to form the 64-bit key.

6.2 Code Transformation

Our ISA extension can be implemented as a compiler extension or via a binary translation approach. For this work, we chose the latter approach. We achieve this by using a proprietary binary transformation framework that is capable of analyzing and manipulating existing binaries. The framework first lifts the code into an intermediate language. On this abstraction level, we analyze the control flow of the program. Furthermore, we are able to add instructions without being constrained by the basic block placement in the memory layout. During code generation, the intermediate language instructions are resolved into a memory layout and adapted to the updated basic block locations. The transformation itself takes place in 3 passes. In the first pass, we add an empty `CHECK` instruction to each basic block. During this pass we identify functions for the return address protection. However, other code or data pointers are not automatically protected and have to be secured manually by the programmer. The second pass simulates the hashing process and identifies possible conflicts, i.e. a basic block is targeting another block which already has an assigned hash resolves them by adding corrections. The third pass performs the actual hashing and inserts the expected values and correction values in the empty placeholder instructions thus creates the finalized program. During this process it may happen, that the framework requires a yet not computed hash, for example, during the pointer encryption and decryption process. Therefore, the framework picks a random known hash for the target location and issues a correction before the decryption takes place. We modified the FSM to cover this case.

Our chosen approach naturally exhibits some weaknesses. For example, during automated program analysis it may not be possible to extract all targets of an indirect call. However, we assume that the benign user wants to protect software. Therefore, the call graph information can be provided as an additional input to the binary transformation framework to resolve such issues. Note that for general-purpose software and a benign user, this problem does not occur if a compiler-based approach is chosen to arm the software with our instruction set extension.

6.3 Software and Hardware Considerations

Until now, we only considered the protection of a single program. However, a complex embedded system may execute multiple processes with shared libraries and interrupts. Therefore, some modifications would have to be done to the software and hardware. In case a (real-time) operating system is used, the context switch has to be altered to include the internal hash register, as the hash state differs for each program. To increase the security, the processor secret may also be different for each process, which would require saving the key register as well. On the hardware side, the unit responsible for saving the execution state when entering an interrupt also has to store the internal hash register (and key register if desired). Additionally, an encryption and hash unit have to be integrated into the processor. To achieve a high throughput both should include an unrolled implementation, which trades space overhead for a shorter execution time. It should be noted, that the hashing can be performed in parallel instead of being a pipeline stage.

6.4 Security Evaluation

In the following section, we systematically evaluate the security of our proposed solution by discussing each potential attack vector and how our design defends against it.

Glitching Attacks Our hashing approach decreases the success probability of a glitching attack by introducing regular checks into the instruction stream. As it is improbable that the hash will still match the expected value when skipping or glitching a single instruction, the glitch will be detected with a latency of $t - i$ where t is the length of the basic block and i the index of the glitched instruction within the block. To circumvent the detection, the attacker is forced to perform multiple glitches. Additionally, the choice of the hash function can restrict the attacker even further as shown by Werner *et al.* [28], because the subsequent bitflips required to hide a fault may be located far apart making it likely that a check happens before a correction is possible.

Control-Flow Attacks The proposed solution can deter glitching-based control-flow attacks as well as traditional software-based attacks. In the first case, the attacker may try to change the target of a branch by forcing a fall through by glitching the processor state. However, by letting the branch result influence the expected state and by introducing a counter, these kind of attacks are severely hampered. For software-based attacks, the gadget-space is drastically reduced, as the return target must be a basic block entry point. Chaining such relatively long instruction sequences together will induce various side-effects. Furthermore, the hash state must match and is influenced by the executed basic blocks. RISC-V specific ROP attacks demonstrated by Jaloyan *et al.* [14] which abuse the mixing of compressed and uncompressed instructions are also unfeasible, as they alter the hash state. Control-flow bending stays possible, as the hash space is limited. However, by employing our pointer protection, the attacker cannot arbitrarily change the return address. Under the assumption that a encryption gadget does not exist, the attacker cannot create his own protected pointers. Instead, he would require an information leak and a write gadget to possibly exchange the return address for another valid return address. Table jumps form a special case, because all targets share the same hash and our pointer protection cannot applied. Here, special precautions have to be made by the programmer in software. However, arbitrary jumps stay impossible as the target must be designated as a jump target.

Memory Attacks Our extension only partially covers memory attacks, as we do not prevent writing or reading from memory. Instead, by using our pointer protection scheme we can stop buffer overflows from being an attack vector. On its own a buffer overflow or any other writing gadget only present a risk, if the attacker is able to overwrite control-flow critical data.

6.5 Evaluation

To evaluate our solution, we chose the MiBench2 [18] benchmark suite, which contains various workloads that are commonly found on embedded devices. We compiled them using the `-Os` compile flag to creates space efficient code. As we are only interested in overhead produced both in the binary and during the execution, we chose to run our *gem5* implementation using the *Atomic-SimpleCPU* model, which simulates a rudimentary single-issue processor, and the *System Emulation* mode to easily load and execute binaries. The results of our evaluation can be seen in Table 2. On average we observe a size overhead of 20.13% and an execution overhead of 28.00%. We note that the overhead created is highly dependent on the size of the basic blocks, as one instruction is added at least for each basic block with the possibility of an additional correction instruction. Small basic blocks will lead to a high amount of additional instructions, whereas large basic

Benchmark	Size Overhead	Execution Overhead
adcpm_encode	37.50%	43.64%
aes	19.86%	21.87%
blowfish	18.18%	11.60%
crc	30.51%	45.60%
fft	26.32%	21.45%
rsa	21.71%	26.89%
sha	19.12%	24.94%
Average	20.13%	28.00%

Table 2. Size and runtime overhead created by our recommendation when compiling with `-Os`.

blocks only require few additional instructions but may give the attacker the possibility to mask his glitch attempt. Therefore, a trade-off must be found. We measured the average basic block length of the `aes` benchmark using different compiler optimizations and found that `-O1`, `-O2` and `-Os` produce smaller basic blocks with an overage length of 8 instructions. On the contrary, `-O0`, so no optimization, as well as `-O3` produce larger basic blocks with an average size of 17 instructions. Consequently, protecting the `aes` benchmark compiled using `-O3` only results in an size overhead of 9.15% and a execution overhead of 2.72%.

Several works exist which implement some parts of our solution. For example, the authors of ISIS [20] add to a control word to each basic block, state a memory overhead of 12% to 15%. However, ISIS misses some protection mechanisms such as the pointer protection and can also not cover all possible branches. The work by Werner *et al.* [27], is the only work at the time of writing, which tries to achieve a holistic protection against faults. However, their protection is based on multiple approaches which do not build upon each other. A mixture of instruction encryption, branch encoding as well as pointer protection is used to protect against glitches. The authors state an average size overhead of 19.8% for the instruction encryption and an average runtime overhead of 9.1%. The branch encoding occurs an additional size overhead of 2.5%. Lastly the pointer protection costs on average 9.99% in binary size and 6.34% in runtime. Thus, a combination of their approaches results in a higher size overhead compared to our solution, whereas our extension increases the amount of executed instructions. Furthermore, a downside of their approach is the encryption process, which is costly and may lead to unpredictable behaviour in case of a glitch, as the decryption may result in a valid but random instruction.

7 Future Work

In this work, we focused on developing a holistic secure ISA extension and implemented it in a simulated environment to determine its overhead. Future work may explore how to implement our recommendation in hardware and analyze its effectiveness by performing clock and voltage glitches. We want to emphasize that this research direction is especially relevant in the context of silicon root-of-trust where a glitch-induced instruction skip can have severe consequences, i.e. to manipulate a secure boot verification check.

8 Conclusion

The rising popularity of resource-constrained embedded devices in security-relevant areas increases the necessity for efficient and effective countermeasures, in particular to protect against attackers with physical access. In this work, we systematically analyzed the state-of-the-art for fault attacks and software-based attacks on embedded systems. We then discussed defense technique combinations and designed a novel approach to form a small instruction set extension that exhibits effective security against the aforementioned threats. We then implemented our approach using binary translation to arm software with our instruction set extension and then evaluated our approach using *gem5* and *MiBench2*. In summary, our results showed that our extension is a competitive candidate for a holistic secure embedded ISA extension.

Acknowledgments

We would like to thank our anonymous reviewers for their constructive feedback. The work described in this paper has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through the project RAINCOAT (440059533), by the DFG through Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and by the German Federal Ministry of Education and Research (BMBF) through the project FlexKI (01IS22086I).

References

1. ARM: Armv8.5-A Memory Tagging Extension White Paper. ARM (2019)
2. Balasch, J., Gierlichs, B., Verbauwhede, I.: An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In: Breveglieri, L., Guilley, S., Koren, I., Naccache, D., Takahashi, J. (eds.) 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011. pp. 105–114. IEEE Computer Society (2011). https://doi.org/10.1109/FDTC.2011.9, https://doi.org/10.1109/FDTC.2011.9
3. Barry, T., Couroussé, D., Robisson, B.: Compilation of a countermeasure against instruction-skip fault attacks. In: Palkovic, M., Agosta, G., Barenghi, A., Koren, I., Pelosi, G. (eds.) Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC, Prague, Czech Republic, January 20, 2016. pp. 1–6. ACM (2016). https://doi.org/10.1145/2858930.2858931, https://doi.org/10.1145/2858930.2858931
4. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers. IACR Cryptol. ePrint Arch. p. 404 (2013), http://eprint.iacr.org/2013/404

5. Bradbury, A., Ferris, G., Mullins, R.: Tagged memory and minion cores in the lowrisc soc. Memo, University of Cambridge (2014)
6. Burow, N., Zhang, X., Payer, M.: Shining light on shadow stacks. CoRR **abs/1811.03165** (2018), http://arxiv.org/abs/1811.03165
7. Christoulakis, N., Christou, G., Athanasopoulos, E., Ioannidis, S.: HCFI: hardware-enforced control-flow integrity. In: Bertino, E., Sandhu, R.S., Pretschner, A. (eds.) Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016. pp. 38–49. ACM (2016). https://doi.org/10.1145/2857705.2857722, https://doi.org/10.1145/2857705.2857722
8. de Clercq, R., Keulenaer, R.D., Coppens, B., Yang, B., Maene, P., Bosschere, K.D., Preneel, B., Sutter, B.D., Verbauwhede, I.: SOFIA: software and control flow integrity architecture. In: Fanucci, L., Teich, J. (eds.) 2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016. pp. 1172–1177. IEEE (2016), https://ieeexplore.ieee.org/document/7459489/
9. de Clercq, R., Verbauwhede, I.: A survey of hardware-based control flow integrity (CFI). CoRR **abs/1706.07257** (2017), http://arxiv.org/abs/1706.07257
10. Cowan, C.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Rubin, A.D. (ed.) Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998. USENIX Association (1998), https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention
11. Davi, L., Hanreich, M., Paul, D., Sadeghi, A., Koeberl, P., Sullivan, D., Arias, O., Jin, Y.: HAFIX: hardware-assisted flow integrity extension. In: Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015. pp. 74:1–74:6. ACM (2015). https://doi.org/10.1145/2744769.2744847, https://doi.org/10.1145/2744769.2744847
12. De, A., Basu, A., Ghosh, S., Jaeger, T.: Hardware assisted buffer protection mechanisms for embedded RISC-V. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(12), 4453–4465 (2020). https://doi.org/10.1109/TCAD.2020.2984407, https://doi.org/10.1109/TCAD.2020.2984407
13. Fei, Y., Shi, Z.J.: Microarchitectural support for program code integrity monitoring in application-specific instruction set processors. In: Lauwereins, R., Madsen, J. (eds.) 2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007. pp. 815–820. EDA Consortium, San Jose, CA, USA (2007). https://doi.org/10.1109/DATE.2007.364391, https://doi.org/10.1109/DATE.2007.364391
14. Jaloyan, G., Markantonakis, K., Akram, R.N., Robin, D., Mayes, K., Naccache, D.: Return-oriented programming on RISC-V. In: Sun, H., Shieh, S., Gu, G., Ateniese, G. (eds.) ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020. pp. 471–480. ACM (2020). https://doi.org/10.1145/3320269.3384738, https://doi.org/10.1145/3320269.3384738
15. Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., Abu-Ghazaleh, N.B.: SCRAP: architecture for signature-based protection from code reuse attacks. In: 19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013. pp. 258–269. IEEE Computer Society (2013). https://doi.org/10.1109/HPCA.2013.6522324, https://doi.org/10.1109/HPCA.2013.6522324

16. Kim, H., Lee, J., Pratama, D., Awaludin, A.M., Kim, H., Kwon, D.: RIMI: instruction-level memory isolation for embedded systems on RISC-V. In: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020. pp. 34:1–34:9. IEEE (2020). https://doi.org/10.1145/3400302.3415727, https://doi.org/10.1145/3400302.3415727
17. Lowe-Power, J., Ahmad, A.M., Akram, A., Alian, M., Amslinger, R., Andreozzi, M., Armejach, A., Asmussen, N., Bharadwaj, S., Black, G., Bloom, G., Bruce, B.R., Carvalho, D.R., Castrillón, J., Chen, L., Derumigny, N., Diestelhorst, S., Elsasser, W., Fariborz, M., Farahani, A.F., Fotouhi, P., Gambord, R., Gandhi, J., Gope, D., Grass, T., Hanindhito, B., Hansson, A., Haria, S., Harris, A., Hayes, T., Herrera, A., Horsnell, M., Jafri, S.A.R., Jagtap, R., Jang, H., Jeyapaul, R., Jones, T.M., Jung, M., Kannoth, S., Khaleghzadeh, H., Kodama, Y., Krishna, T., Marinelli, T., Menard, C., Mondelli, A., Mück, T., Naji, O., Nathella, K., Nguyen, H., Nikoleris, N., Olson, L.E., Orr, M.S., Pham, B., Prieto, P., Reddy, T., Roelke, A., Samani, M., Sandberg, A., Setoain, J., Shingarov, B., Sinclair, M.D., Ta, T., Thakur, R., Travaglini, G., Upton, M., Vaish, N., Vougioukas, I., Wang, Z., Wehn, N., Weis, C., Wood, D.A., Yoon, H., Zulian, É.F.: The gem5 simulator: Version 20.0+. CoRR **abs/2007.03152** (2020), https://arxiv.org/abs/2007.03152
18. Mibench2 (2022), https://github.com/impedimentToProgress/MiBench2
19. Ohlsson, J., Rimén, M., Gunneflo, U.: A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog. In: Digest of Papers: FTCS-22, The Twenty-Second Annual International Symposium on Fault-Tolerant Computing, Boston, Massachusetts, USA, July 8-10, 1992. pp. 316–325. IEEE Computer Society (1992). https://doi.org/10.1109/FTCS.1992.243569, https://doi.org/10.1109/FTCS.1992.243569
20. Rodríguez, F., Campelo, J.C., Serrano, J.J.: A watchdog processor architecture with minimal performance overhead. In: Anderson, S., Bologna, S., Felici, M. (eds.) Computer Safety, Reliability and Security, 21st International Conference, SAFECOMP 2002, Catania, Italy, September 10-13, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2434, pp. 261–272. Springer (2002). https://doi.org/10.1007/3-540-45732-1_26, https://doi.org/10.1007/3-540-45732-1_26
21. Rodríguez, F., Serrano, J.J.: Control flow error checking with ISIS. In: Yang, L.T., Zhou, X., Zhao, W., Wu, Z., Zhu, Y., Lin, M. (eds.) Embedded Software and Systems, Second International Conference, ICESS 2005, Xi'an, China, December 16-18, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3820, pp. 659–670. Springer (2005). https://doi.org/10.1007/11599555_63, https://doi.org/10.1007/11599555_63
22. Savry, O., El-Majihi, M., Hiscock, T.: Confidaent: Control flow protection with instruction and data authenticated encryption. In: 23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020. pp. 246–253. IEEE (2020). https://doi.org/10.1109/DSD51259.2020.00048, https://doi.org/10.1109/DSD51259.2020.00048
23. Security, Q.P.: Pointer Authentication on ARMv8.3 - Design and Analysis of the New Software Security Instructions. Qualcomm Technologies, Inc. (January 2017)
24. Selmke, B., Hauschild, F., Obermaier, J.: Peak clock: Fault injection into pll-based systems via clock manipulation. In: Chang, C., Rührmair, U., Holcomb, D.E., Schaumont, P. (eds.) Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, ASHES@CCS 2019, London, UK, November 15, 2019. pp. 85–94. ACM (2019). https://doi.org/10.1145/3338508.3359577, https://doi.org/10.1145/3338508.3359577

25. Shanbhogue, V., Gupta, D., Sahita, R.: Security analysis of processor instruction set architecture for enforcing control-flow integrity. In: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2019, June 23, 2019. pp. 8:1–8:11. ACM (2019). https://doi.org/10.1145/3337167.3337175, https://doi.org/10.1145/3337167.3337175
26. Spensky, C., Machiry, A., Burow, N., Okhravi, H., Housley, R., Gu, Z., Jamjoom, H., Kruegel, C., Vigna, G.: Glitching demystified: Analyzing control-flow-based glitching attacks and defenses. In: 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021. pp. 400–412. IEEE (2021). https://doi.org/10.1109/DSN48987.2021.00051, https://doi.org/10.1109/DSN48987.2021.00051
27. Werner, M., Schilling, R., Unterluggauer, T., Mangard, S.: Protecting RISC-V processors against physical attacks. In: Teich, J., Fummi, F. (eds.) Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019. pp. 1136–1141. IEEE (2019). https://doi.org/10.23919/DATE.2019.8714811, https://doi.org/10.23919/DATE.2019.8714811
28. Werner, M., Wenger, E., Mangard, S.: Protecting the control flow of embedded processors against fault attacks. In: Homma, N., Medwed, M. (eds.) Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers. Lecture Notes in Computer Science, vol. 9514, pp. 161–176. Springer (2015). https://doi.org/10.1007/978-3-319-31271-2_10, https://doi.org/10.1007/978-3-319-31271-2_10
29. Wilken, K.D., Shen, J.P.: Continuous signature monitoring: Efficient concurrent-detection of processor control errors. In: Proceedings International Test Conference 1988, Washington, D.C., USA, September 1988. pp. 914–925. IEEE Computer Society (1988). https://doi.org/10.1109/TEST.1988.207880, https://doi.org/10.1109/TEST.1988.207880
30. Witteman, M., Oostdijk, M.: Secure application programming in the presence of side channel attacks. In: RSA conference. vol. 2008 (2008)
31. Woodruff, J., Joannou, A., Xia, H., Fox, A.C.J., Norton, R.M., Chisnall, D., Davis, B., Gudka, K., Filardo, N.W., Markettos, A.T., Roe, M., Neumann, P.G., Watson, R.N.M., Moore, S.W.: CHERI concentrate: Practical compressed capabilities. IEEE Trans. Computers **68**(10), 1455–1469 (2019). https://doi.org/10.1109/TC.2019.2914037, https://doi.org/10.1109/TC.2019.2914037
32. Woodruff, J., Watson, R.N.M., Chisnall, D., Moore, S.W., Anderson, J., Davis, B., Laurie, B., Neumann, P.G., Norton, R.M., Roe, M.: The CHERI capability model: Revisiting RISC in an age of risk. In: ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014. pp. 457–468. IEEE Computer Society (2014). https://doi.org/10.1109/ISCA.2014.6853201, https://doi.org/10.1109/ISCA.2014.6853201
33. Yuce, B., Ghalaty, N.F., Deshpande, C., Patrick, C., Nazhandali, L., Schaumont, P.: FAME: fault-attack aware microprocessor extensions for hardware fault detection and software fault response. In: Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016. pp. 8:1–8:8. ACM (2016). https://doi.org/10.1145/2948618.2948626, https://doi.org/10.1145/2948618.2948626
34. Zhu, G., Tyagi, A.: Protection against indirect overflow attacks on pointers. In: Cole, J.L., Wolthusen, S.D. (eds.) Proceedings of the Second IEEE International Workshop on Information Assurance (IWIA'04), April 8-9, 2004, Charlotte, North

Carolina, USA. pp. 97–106. IEEE Computer Society (2004). https://doi.org/10.1109/IWIA.2004.1288041, https://doi.org/10.1109/IWIA.2004.1288041