Protecting the Control Flow of Embedded Processors against Fault Attacks

Mario Werner[1], Erich Wenger[2,⋆], and Stefan Mangard[1]

[1] Graz University of Technology, Austria
{mario.werner, stefan.mangard}@iaik.tugraz.at
[2] Infineon Technologies AG, Munich, Germany
erich.wenger@infineon.com

Abstract. During the last two decades, most of the research on fault attacks focused on attacking and securing intermediate values that occur during the computation of cryptographic primitives. However, also fault attacks on the control flow of software can compromise the security of a system completely. Fault attacks on the control flow can for example make a system branch to an administrative function directly or make it bypass comparisons of redundant computations. Security checks based on comparing redundant computations are for example commonly used to secure PIN checks and implementations of block ciphers against fault attacks.

Although control-flow integrity is of crucial importance to secure a system against fault attacks, so far there exist only very few proposals for countermeasures. This article addresses this gap and presents an efficient hardware-supported technique that allows to maintain control-flow integrity in the setting of fault attacks. The technique is based on so-called generalized path signatures, which have initially been introduced in the context of soft errors. We present a prototype implementation for a Cortex-M3 microprocessor and corresponding compiler extensions in LLVM. Our implementation, which increases the processor size by merely 6.4 %, detects *every* fault on the instruction-stream with 99.9 % probability within 3 cycles. The runtime overhead of the protected applications ranges from 2 % to 71 %.

Keywords: control-flow integrity, fault attacks, countermeasures.

1 Introduction

Fault attacks are a very active field of research since the seminal publication of the so-called Bellcore attack [3] in 1997. Today, there exist published fault attacks on almost all commonly used cryptographic primitives. Unless countermeasures are implemented, these attacks allow to reveal the secret key by observing only few outputs of faulted executions of the cryptographic primitive. A comprehensive overview of fault attacks and countermeasures for cryptographic primitives can be found in [5].

⋆ This research has been conducted while Erich Wenger was employed at Graz University of Technology.

However, while most of the research on fault attacks focuses on attacking and securing cryptographic primitives, it is important to point out that securing a cryptographic primitive is not sufficient to secure a system. For example, the Xbox 360 has not been hacked because of a fault attack on a cryptographic primitive. It has been attacked successfully because it was possible to use a glitch on the reset line to make the system bypass the signature check of the loaded software [4]. In case of such attacks on the control flow of the executed software, often a single successful fault induction is sufficient to compromise the security of a system completely (e.g. by branching to an administrative function, by obtaining root privileges, or by skipping all kinds of security checks). Attacks on the control flow also allow to bypass certain countermeasures for cryptographic computations. In [12] for example, techniques for multiple fault inductions are discussed to first induce a fault in a cryptographic computation and then to bypass the comparison with a redundant computation.

So far, there exist only very few publications that address the challenge of securing the control flow of software against fault attacks. Examples of hardware-supported approaches are [2], [8] and [9]. However, these works focus on securing basic blocks and lead to a significant overhead in terms of code size when protecting the entire control-flow graph. In [6], a software approach for securing the control flow is presented. Yet, this approach only allows to detect integrity violations at a coarse level of granularity. It does not allow detecting all modified, missing, or repeated instructions with certainty.

The present article addresses the current lack of efficient countermeasures to secure the control flow against fault attacks. We present an efficient hardware-supported technique to provide strong security for the integrity of the control flow. Our technique builds upon generalized path signature analysis that has been introduced in the context of soft errors by Wilken and Shen [10,11].

In this article, we define the requirements for fault detection in the setting of fault attacks and adapt the scheme of Wilken and Shen accordingly. Furthermore, we present an implementation of the resulting countermeasure using state-of-the-art hardware (ARM Cortex-M3) and software (LLVM compiler infrastructure). To the best of our knowledge, this work is the first to actually implement a control-flow integrity scheme based on general path signature analysis. Our prototype implementation, which increases the processor size by 6.4 %, detects *every* fault on the instruction-stream with 99.9 % probability within three cycles. The runtime overhead of the protected applications ranges from 2 % to 71 %. The overhead for implementations of cryptographic primitives is very low because such software typically has a low number of conditional branches. The handling of conditional branches causes the main part of the overhead.

The remainder of this article is organized as follows. Section 2 gives an introduction on control-flow integrity and the existing work of Wilken and Shen. We adapt the scheme to the setting of fault attacks in Section 3. Section 4 presents our prototype implementation and Section 5 the evaluation results. Finally, the work concludes in Section 6.

2 Control-Flow Integrity in Fault-Tolerant Computing

The detection of faults in the control flow of a program requires to include redundant information about the control flow into the program. The concept of generalized path signature analysis (GPSA) by Wilken and Shen [10,11] is a very efficient technique to add this redundancy. In the following subsections, we first define the problem of control-flow integrity and then discuss the concept of GPSA.

2.1 Control-Flow Integrity

The control flow of a program refers to the order in which its instructions, branches, loops, and function calls have to be executed. Two types of instructions can be distinguished in this context. First, there are sequential instructions, like arithmetic and memory operations, which only have indirect influence on the execution sequence. They are executed in strictly sequential order and have exactly one subsequent instruction. Second, there are control-flow instructions, like `branch` and `call`, which alter the execution sequence directly. Control-flow instructions have one or more subsequent instructions and can select which one is executed next.

A program is typically structured into code fragments which consist out of an arbitrary number of sequential instructions (zero or more) followed by up to one control-flow instruction. Such fragments are denoted as basic blocks. A basic block is a strictly sequential piece of code which can only be entered at the first and exited after the last instruction. All basic blocks of a program form the so-called control-flow graph (CFG). The edges in a CFG are always directed and visualize in which way the control flow can be transferred from one basic block to another. Ensuring control-flow integrity (CFI) during the execution of a program means that all instructions in a basic block are executed by the processor as defined in the original program (*i.e.* no instructions are skipped or altered) and that no new connections are added to the control-flow graph (*i.e.* no other branches are done than those defined at compilation time).

Control-flow integrity does not include the protection of the decision which path is taken in a CFG. This requires protecting the integrity of the data that is used for the decision. However, it is important to note that data integrity cannot be achieved without control-flow integrity. CFI is the basis for further countermeasures, like data integrity. For example multiple computations and comparisons can be done to ensure data integrity and the techniques for CFI make sure that all these operations are indeed executed by the processor.

2.2 Derived Signatures

Derived signatures are a common technique in fault-tolerant computing to detect violations of the integrity of the control flow. The basic idea of a derived signature is to add a small piece of hardware to the processor executing the software that should be protected. Upon the execution of each instruction, the hardware updates a checksum based on the executed instruction and the corresponding control signals of the decoder. In the literature on CFI in fault-tolerant computing, such a checksum is called " derived signature". It is important to note that it is not a cryptographic signature. Nevertheless, in order to be consistent with the existing literature, we also use the term derived signature to denote a checksum that is calculated in hardware based on a sequence of executed instructions.

In order to check such a derived signature when a program is executed, it is necessary to have corresponding reference values. Derived signatures depend on the executed instructions and the initial value of the signature. As both are known at compilation time, reference values can be calculated when a program is created. Typically, the reference values are embedded into the program by instrumenting the binary, either during compilation or in a post-processing step.

Derived signatures can for example be checked at the end of every basic block. This is shown in Figure 1a. The figure shows a control-flow graph with six basic blocks, labelled %1 to %6 that include a while loop. At the end of each basic block a signature check is done and therefore a reference value for each basic block has to be added to the program. This is for example done in [2], [8] and [9]. However, this leads to a significant overhead, which can be avoided when using generalized path signatures. Furthermore, there is no protection for the connections of the basic blocks.

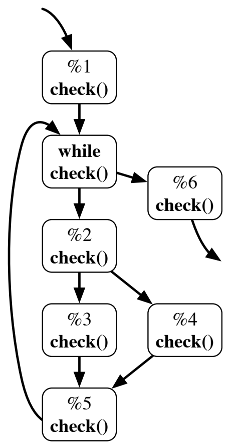
2.3 Generalized Path Signature Analysis

In [7], the so-called path signature analysis (PSA) has been introduced. PSA checks the integrity not only for a basic block, but along paths through a control-flow graph. This significantly reduces the overhead. Wilken and Shen in [10,11] extended PSA into generalized path signature analysis (GPSA) in order to optimize the overhead.

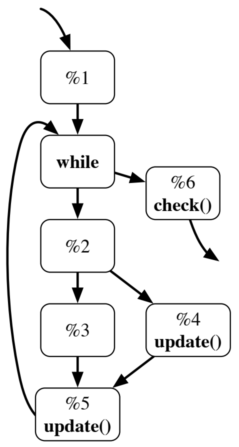
The basic idea of GPSA is to insert signature updates into the program code in such a way that independent of the used paths in a CFG, the signature value at a given instruction is always the same. This idea is illustrated in Figure 1b. At the end of basic block %4, there is an update that makes sure that the signature at the beginning of %5 is the same independent of the fact whether it is reached via %3 or %4. The update at the end of %5 ensures that the signature at the beginning of the while loop is independent of the fact whether it is reached via %1 or %5. The values that need to be stored in the program code to do the updates are called justifying signatures [10] and they are calculated at compilation time—just like the reference values for the checks.

The concept of GPSA optimizes the number of total justifying signatures in a CFG and can also be extended to protect function calls. In case of function calls, there is an additional justifying signature necessary for each function call. For details, please refer to [10].

GPSA does not require to have a check in every basic block and allows to place signature checks at arbitrary positions in the program. These checks are denoted as vertical signature checks. At minimum, it is necessary to insert one signature check at the end of the program as it is done in Figure 1b. Depending



(a) Basic block checking



(b) GPSA updates + checking

Fig. 1: Signature based checking methodologies.

on the application, a trade-off has to be made between between runtime and memory overhead on the one hand and the detection latency on the other hand.

2.4 Continuous-Signature Monitoring

Wilken and Shen proposed continuous-signature monitoring (CSM) as an alternative concept to the manual placement of signature checks and to solve the latency problem of vertical signature checks. The idea of CSM is to check the signature, or at least parts of it, on every executed instruction. Implementing CSM on top of GPSA is therefore as simple as checking h bits of the $|S|$ bit runtime signature on every executed instruction.

It has been proposed to use spare bits in the instruction encodings or to embed reference information into the error-correction/detection bits of the memory system. However, these approaches are not applicable to most modern processor architectures given their dense instruction encodings and the lack of error detecting memory.

3 Control-Flow Integrity in the Setting of Fault Attacks

Fault attack detection is per se very similar to the detection of soft errors. The main difference between the two is the fault model. Soft errors occur randomly at a low frequency. Fault attacks on the other hand can be very controlled. When comparing different checksums for derived signatures, it is therefore important to not only look at average detection probabilities but to also keep the worst case scenario in mind.

This section elaborates on the required properties that are needed in order to make the schemes of Wilken and Shen ready for fault attacks. We define functional requirements for both, the signature and the update function, which make single faulty instructions detectable with certainty. We further show that the actual function selection has an huge impact on the detection capabilities. The best of the evaluated functions can detect up to 7 faulty bits in the instruction stream across two cycles with certainty.

3.1 Signature Function Selection

The calculation of derived signatures can be modeled using a compression function f which is used in a Merkle-Damgård-like mode of operation. The next signature $S_{j+1} = f(S_j, I_j)$ is calculated based on the preceding signature S_j and the current instruction I_j. Collisions across multiple iterations of the signature function are unavoidable given that the signature value S has fixed size. However, choosing a signature function with specific properties can at least provide certain worst-case guarantees.

Functional Requirements. The signature function f needs the following properties in order to make a single faulty instruction $I_j \oplus \Delta_{I_j}$ detectable with certainty, independent of the actual error Δ_{I_j} and the number of faulty bits $HW(\Delta_{I_j})$.

– *Reliability*: Every error in the instruction stream ($\Delta_{I_j} \neq 0$) has to result in a signature error ($\Delta_{S_{j+1}} \neq 0$) given that the original signature was correct ($\Delta_{S_j} = 0$). Note that this requirement can only be fulfilled if $|S| \geq |I|$.

$$S_{j+1} \oplus \Delta_{S_{j+1}} = f(S_j, I_j \oplus \Delta_{I_j}), \quad \forall \Delta_{I_j} \neq 0 \rightarrow \Delta_{S_{j+1}} \neq 0 \tag{1}$$

– *Error preservation*: An error, absorbed into the signature $S_j \oplus \Delta_{S_j}$, must not be eliminated by an error-free sequence of inputs ($\Delta_{I_j} = 0$). This requirement allows to arbitrarily delay the checking of a signature. Consequently, the number of necessary signature checks can be reduced.

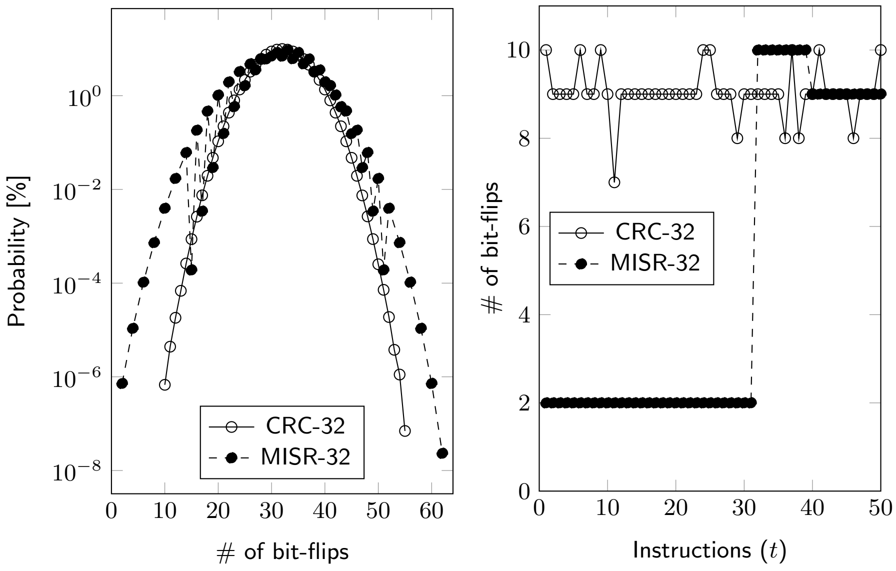
$$S_{j+1} \oplus \Delta_{S_{j+1}} = f(S_j \oplus \Delta_{S_j}, I_j), \quad \forall \Delta_{S_j} \neq 0 \rightarrow \Delta_{S_{j+1}} \neq 0 \tag{2}$$

– *Non associativity*: The order in which instructions I_j, I_k are absorbed by f must have an influence on the resulting signature value.

$$\forall I_j \neq I_k \rightarrow f(f(S_j, I_j), I_k) \neq f(f(S_j, I_k), I_j) \tag{3}$$

– *Invertibility*: Depending on the concrete implementation of the scheme, invertibility may also be a requirement. The signature function should therefore be invertible in S given S_{j+1} and I_j. Our implementation for example uses this property to be able to place signature updates at arbitrary places along a path through the CFG. A different implementation, which enforces that signature updates are only performed at merging points in the CFG, would be able to cope without this property.

$$S_j = f^{-1}(S_{j+1}, I_j), \quad \forall S_{j+1}, \forall I_j \tag{4}$$



(a) Probability density function for $q(j, 1)$. (b) Min. # of bit-flips for collision.

Fig. 2: Comparison between CRC-32 and MISR-32.

Choosing the Signature Function. Classical choices for checksums in the setting of fault-tolerant computing are cyclic redundancy checks (CRCs) and multiple-input signature registers (MISRs) with various polynomials. MISRs as well as CRCs fulfill the mentioned requirements. However, they are not equally suited when fault attacks with high control over the injected fault are considered.

For the evaluation of different signature functions, we evaluated the number of bit-flips required to introduce a fault on one instruction Δ_{I_j} and to compensate it with a fault on a subsequent instruction $\Delta_{I_{j+t}}$. The sum of the bit-flips required for both faults $q(j,t) = HW(\Delta_{I_j}) + HW(\Delta_{I_{j+t}})$ is a measure for the attack complexity. The quality function q has been chosen in this way to take into account that exact knowledge of the injected fault is needed in order construct and subsequently inject the compensating fault. Average as well as worst-case performance is important when fault attacks are considered.

A comparison between the signature functions CRC-32 and MISR-32 (identical polynomial) based on the probability density function of $q(j,t)$ at $t = 1$ is shown in Figure 2a. CRC-32 as well as MISR-32 have an expected number of bit-flips of 32. The expected value for $q(j,t)$ in general is identical to the degree of the reduction polynomial for both MISR and CRC codes. Performance in the average case is therefore identical which makes them equally suited for soft error detection.

The worst-case performance on the other hand is different. The comparison in Figure 2b ($min(q)\ \forall \Delta_{I_j}, \forall \Delta_{I_{j+t}}$) shows that the CRC-32 is superior to the MISR-32 regarding worst-case performance. The used CRC enforces that at least

7 bit-flips are needed in order to construct a collision. The MISR on the other hand can already be defeated using 2 bit-flips within the first 31 instructions. This weakness is caused by the simple structure of the MISRs which makes them not suited as signature functions in the fault attack context. A more extensive comparison between various polynomials regarding worst-case performance can be found in Table 1.

Table 1: Performance ($min(q)$ $\forall t = [1, 50], \forall \Delta_{I_j}, \forall \Delta_{I_{j+t}}$) of different polynomials. The polynomials are given in reversed representation.

Type	Polynomial	$min(q)$	t
CRC-8	`0xAB`	2	11
CRC-16-ARINC	`0xD405`	4	10
CRC-16-CCITT	`0x8408`	4	1
CRC-16-CDMA2000	`0xE613`	4	32
CRC-16-DECT	`0x91A0`	2	15
CRC-16-T10-DIF	`0xEDD1`	4	7
CRC-16-DNP	`0xA6BC`	2	9
CRC-16-IBM	`0xA001`	4	1
CRC-32	`0xEDB88320`	7	11
CRC-32C (Castagnoli)	`0x82F63B78`	8	2
CRC-32K (Koopman)	`0xEB31D82E`	6	34
CRC-32Q	`0xD5828281`	8	2
MISR-32	`0xEDB88320`	2	1

3.2 Update Function Selection

The second function which is required in GPSA and CSM implementations is the so-called update function. This function is needed in order to balance the various paths through the control-flow graph. The update function u calculates the next signature $S_{j+1} = u(S_j, J_j)$ based on the preceding signature S_j and a justifying signature constant J_j. The update function has to fulfill the following requirements in order to be usable for GPSA.

- *Full control*: All possible signature values S_{j+1} have to be constructible given an arbitrary S_j and a justifying signature J_j. Note that, the size of J must be larger or equal to S ($|J| \geq |S|$) to modify each bit in S.

$$S_{j+1} = u(S_j, J_j), \quad \forall S_{j+1}, \forall S_j, \exists J_j \tag{5}$$

- *Error preservation*: An error, absorbed into the signature $S_j \oplus \Delta_{S_j}$, must not be eliminated by an error-free justifying signature ($\Delta_{J_j} = 0$). It would otherwise not be possible to arbitrarily delay the actual checking.

$$S_{j+1} \oplus \Delta_{S_{j+1}} = u(S_j \oplus \Delta_{S_j}, J_j), \quad \forall \Delta_{S_j} \neq 0 \rightarrow \Delta_{S_{j+1}} \neq 0 \tag{6}$$

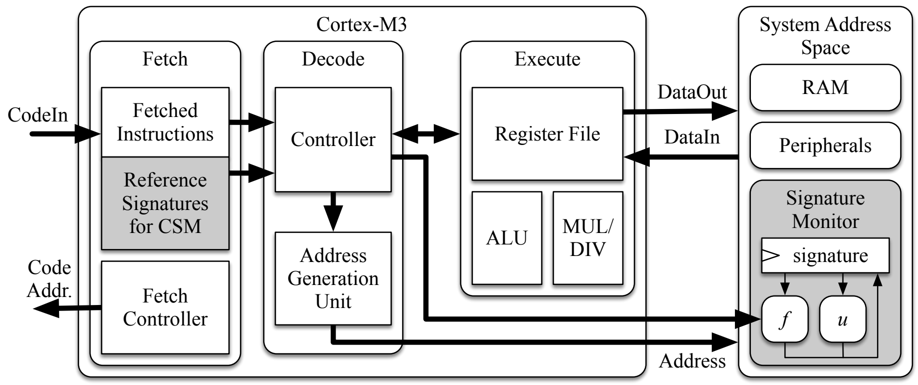


Fig. 3: Simplified Cortex-M3 architecture with grey-shaded modifications.

– *Invertibility*: Given S_{j+1} and S_j, it must be possible to efficiently compute the justifying signature J_j.

$$J_j = u^{-1}(S_j, S_{j+1}), \quad \forall S_j, \forall S_{j+1} \tag{7}$$

A simple function which fulfills all those requirements is the binary `xor` function.

4 Prototype Implementation

We implemented GPSA and CSM on the basis of a state-of-the-art microprocessor architecture (an ARM Cortex-M3) and modern compiler technology (LLVM). The resulting implementation supports all C language features and common programming practices. Our prototype implementation is therefore not only a theoretic construct, but practically usable. The implementation supports separate compilation of C files and enables the use of static libraries. It also allows to randomize the signature values of identical programs (diversity) on different devices. This makes it harder to extend attacks against an individual towards multiple devices.

In this section, we discuss the necessary hardware modifications (which are minimal), the necessary modifications of the to-be-protected software, and elaborate on the modifications of the toolchain.

4.1 Hardware Architecture

The presented implementation is based on the ARM Cortex-M3 microprocessor architecture. Its performance-to-energy ratio makes this processor an interesting candidate for many embedded application areas, including smart-card applications. Hence, they are often used in malicious environments. The processor uses

3 pipeline stages to implement the ARMv7-M instruction set that supports both Thumb and Thumb-2 instructions.

As depicted in Figure 3, the Cortex-M3 was extended with a memory-mapped signature monitor which is tightly integrated into the design. This monitor automatically computes $|S| = 32$-bit signatures absorbing the 16–32-bit large Thumb and Thumb-2 instructions I_j. The CRC-32C code has been implemented as signature function based on the analysis presented in Section 3.1. Via the memory interface, the monitor enables the CPU to perform signature updates, signature replacements, and signature assertions. Assertion failures can either trigger an interrupt or reset the system.

To support the automatic computation of derived signatures, the CPU only had to be modified to forward the currently executed instruction to the monitor. To perform continuous signature monitoring, the fetch unit of the processor was modified. The signature bits are stored in a block at the end of the program. The base address of this signature block has been embedded into the interrupt vector table, similar like the initial stack pointer. At start-up, the base address is automatically initialized. During run-time, the fetch unit always loads the instructions in combination with the reference values. An instruction is only forwarded to the decode stage, once both the instruction and the reference value are valid.

4.2 Source Code Modifications

All signature modifications are performed in software, which in turn are monitored by the derived-signature monitor in hardware. Performing the necessary software transformations manually is a challenging and error prone task. It is clearly favorable to automatically perform the transformations within the toolchain, which makes the whole instrumentation process transparent for the programmer. Consequently, modifications of the application C source code are minimal. In the best case, a to-be-protected software does not have to be modified at all.

The programmer can insert vertical signature checks in the form of `assert_signature()` function calls into critical sections of the program. All remaining work is performed by the compiler which automatically replaces these function calls with actual signature checks. The use of function calls for the annotation has the advantage that `clang`, LLVM's C front end, can be used without any modification.

Assembly code on the other hand requires a little more work (as usual). The programmer has to place signature updates by hand when branches, loops, and function calls are encountered. However, no actual derived signature calculation has to be performed by the programmer. Additionally, if the programmer forgets a signature update, the toolchain will automatically notify her.

4.3 Software Modifications

Related work [1], [8], [9], [13] usually performs the software transformations either during compilation or by applying a dedicated post-processing tool after linking. In this work, both techniques are combined in order to generate a protected executable. The compiler is responsible to insert signature updates based on GPSA and to insert signature assertions. A post-processing tool consecutively computes the derived signatures and patches the executable with signature update and reference values.

LLVM Compiler Modifications. The compiler has been built using the LLVM compiler infrastructure which already has great support for the targeted ARMv7-M architecture.

A machine function pass has been added to the ARM back-end in order to perform the following transformations:

- *Insertion of asserts*: Every call to the `assert_signature()` function is replaced by an actual vertical signature check. A signature check is performed as a memory-write operation of the expected signature to a certain predefined monitor address and is composed of three instructions. (LOAD address, LOAD value, STORE value)
- *Insertion of signature updates*: Signature updates are inserted to make the runtime signature independent of the executed path through the control-flow graph. The placement of signature updates is performed efficiently by computing the spanning tree of a function's undirected control-flow graph. Signature updates are, similar to checks, a write of the justifying signature to the memory mapped monitor.

The smart placement of the machine function pass in the optimization pipeline allows us to reuse much of the original compiler's functionality and therefore benefit from the available optimizations as well. Register allocation is for example still handled using stock LLVM functionality.

An additional component which had to be adapted is the run-time library. The compiler relies on its functions for standard operations (e.g., clearing memory) or to perform computations which are not natively supported by the processor (e.g., floating-point operation). It was therefore necessary to instrument this library with justifying signature updates in order to generate a working GPSA-hardened program.

Post Processing Tool. As a result, the compiler generates a binary with all necessary signature updates/assertions that still lacks the correct signature constants. The signature values can only be computed once the program is linked and all instructions have been finalized. The compiler never has access to this information in a traditional separate-compilation design-flow. We therefore perform the derived signature calculation using a post-processing tool.

A recursive disassembling [13] approach was used to recover the control flow and the location of the signature constants within this post-processing tool. LLVM's disassembling machinery simplifies this step considerably. Based on the control flow it is possible to identify the constant pools (aka constant islands) in the binary. Tracking the monitor's addresses using data-flow analysis techniques consecutively reveals the location of the instructions which modify the signature values.

The actual calculation of the derived signatures relies on all this recovered information. The signature values are computed by initializing each function with a random initial signature, and consequently flooding the control-flow graph of each function. As a result, all justifying signatures, assertion constants, and reference signatures for the CSM are embedded into the executable.

Another feature of the post-processing tool is its static code analysis functionality of the binary. Only correctly instrumented binaries pass the derived signature calculation. Error messages notify a programmer about wrongly instrumented assembly code.

5 Evaluation

As this is the first published, practical implementation of both GPSA and CSM in the context of fault attacks, we are excited to report performance results based on qualitative characteristics as well as practical benchmarks.

5.1 Error-detection Coverage

Based on the previously stated requirements on the signature functions, every single fault on the instruction stream changes the runtime signature with certainty. Using vertical checks, any runtime-signature error can be detected. On the contrary, CSM checks h bits of the runtime signature per cycle. Therefore, the probability to detect an error is $1 - 2^{-h}$. As any error propagates within the signature register, the probability to detect an error is way beyond 99.9 % after 3 checks of $h = 4$ bits.

If an attacker targets two instructions, she could possibly hide the error by colliding the signature value. It was shown in Section 3.1 that the attacker has to flip 32 bits on average or 8 bits in his best case when a CRC-32C is used as signature function. Even using advanced attack setups, the probability for introducing a fault with precise bit-flips across multiple cycles is very low.

5.2 Error-detection Latency

An error can only be detected at the time of the vertical signature check when GPSA is used without CSM. It is up to the programmer to insert these vertical checks next to the critical pieces of code. This allows to perform very controlled checks and consecutively reduces overhead. However, it is possible that, due to bad check placement, vertical signature checks by itself detect an error once it already has been exploited.

CSM solves this problem given that it checks parts of the signature register after every executed instruction. With an increasing probability any error is detected after a few iterations.

5.3 Monitor Complexity

One of our design goals was to only introduce minimal hardware overhead. All operations beside derived signature calculation are performed entirely in software. We evaluated the monitor complexity after synthesis for UMC's 130nm Low Leakage process using Cadence 2009 tools. The standard cell library for this process comes from Faraday. Without the monitor, our Cortex-M3 is 36,957 GE large. Adding the monitor for GPSA increases the size of the processor by only 1,469 GE, respectively by less than 4 %. Adding support for CSM additionally increases the size of the fetch unit which results in a total core size of 39,319 GE. The modifications to support GPSA and CSM therefore are minimal and account to merely 6.4 % hardware overhead.

5.4 Memory Overhead and Processor-Performance Loss

Memory overhead and processor-performance loss highly depend on the executed program. These characteristics are mainly determined by the number of branches, function calls, and vertical signature checks.

Qualitatively speaking, a single signature update costs around 10 bytes of memory and 6 cycles in our software-centered implementation. A function call costs around 14 bytes of memory and 10 cycles. Using CSM, the introduced redundancy is proportional to the size of the code within the `text` section of the executable. For $h = 4$ per 16-bit Thumb instruction, up to 25 % of redundant NVM has to be added.

For a quantitative, empirical evaluation, we tested multiple programs: a coremark benchmark (one iteration), an AES-256 roundtrip (encryption followed by decryption with check), and a 160-bit elliptic curve cryptography (ECC) example performing a scalar multiplication with optional assembly-optimized finite-field arithmetic. The coremark benchmark has been optimized for speed (`-O2`) given that this yields the best performance. The crypto algorithms have been optimized for size (`-Os`). Additionally, link-time garbage collection (`-ffunction-sections -fdata-sections` and `-Wl,-gc-sections`) has been used to preserve only the absolutely necessary code and data segments. A synthesizeable VHDL model of the hardware, evaluated using Cadence NC Sim, has been used to execute the benchmarks.

The raw numbers and the relative overhead in terms of runtime as well as RAM and NVM size are summarized in Table 2. The evaluation was performed in two steps. First, our GPSA implementation is compared against the unmodified LLVM backend which is used as baseline. Second, CSM is compared with the GPSA version given that it extends GPSA's checking capabilities.

RAM. The RAM overhead of GPSA is below 10 % in all eveluated programs. For coremark it is even merely 3 %. This overhead is solely a side effect of the increased register pressure during function calls. The additional live variables force the compiler to spill more values and therefore slightly increase the memory usage on the stack. Using CSM on top of GPSA introduces no additional RAM overhead given that the code itself stays absolutely unchanged.

NVM. Overhead on the NVM side ranges from 29 % for the AES test case to 79 % for ECC. This overhead is composed of the actual signatures (justifying + reference) and the added code for signature updates and vertical checks. In this software-centered implementation, the majority of the overhead is code. The signatures account for 25 % NVM overhead at most.

The NVM overhead of CSM over GPSA on the other hand is purely signature based. Only minor optimization potential remains.

Runtime. The most remarkable figure in this evaluation is probably the runtime overhead. The overhead of GPSA ranges from 2 % for optimized ECC to 57 % for

Table 2: Empirical Results for GPSA and CSM regarding RAM, NVM, and runtime overhead. Additionally, the NVM overhead solely for justifying and reference signatures is given.

Program	**RAM** Byte	**NVM** Byte	**Runtime** Cycle	**Justifying Sigatures**	**Reference Sigatures**
Baseline					
Coremark	2,444	9,384	547,294	—	—
AES-256	248	3,212	48,581	—	—
ECC	444	4,036	4,251,697	—	—
ECC w/ ASM	400	4,824	2,836,180	—	—
Overhead of GPSA (Relative to Baseline)					
Coremark[a]	2.3 %	69.0 %	56.7 %	23.5 %	0.1 %
AES-256[b]	9.6 %	29.0 %	36.7 %	10.8 %	0.5 %
ECC[c]	9.0 %	78.9 %	33.3 %	24.5 %	0.3 %
ECC w/ ASM[c]	8.0 %	53.5 %	1.9 %	16.3 %	0.2 %
Overhead of CSM with $h = 4$ bit (Relative to GPSA)					
Coremark[a]	—	22.2 %	8.9 %	—	22.2 %
AES-256[b]	—	19.3 %	6.5 %	—	19.3 %
ECC[c]	—	21.9 %	7.6 %	—	21.9 %
ECC w/ ASM[c]	—	22.6 %	0.4 %	—	22.6 %

[a] One vertical signature check before and one after the benchmark.
[b] One vertical signature check after every round of AES.
[c] One vertical signature check after every processed bit of the scalar.

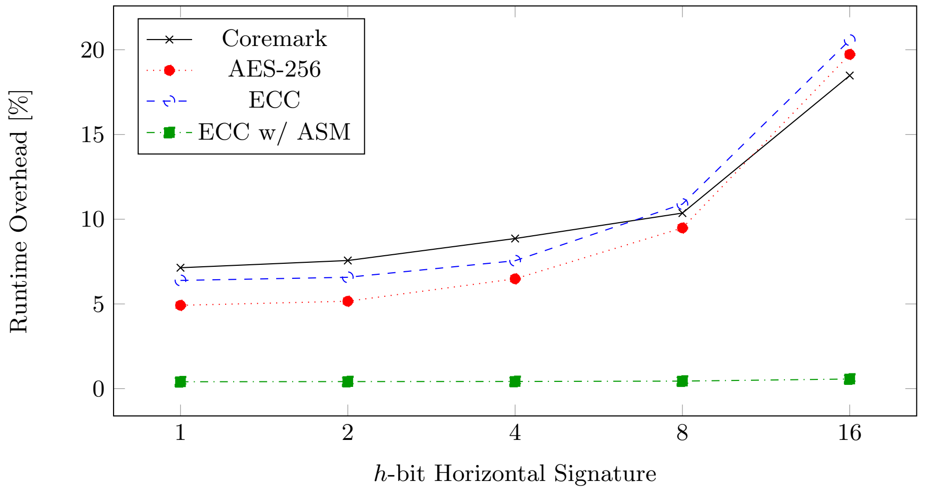


Fig. 4: Runtime overhead of CSM with different horizontal signature sizes (h-bit). (Relative to GPSA)

coremark. The software-centered approach taken in this implementation is again one of the reasons for these high values. Each GPSA operation takes between 6 and 10 cycles. Adding more hardware support could bring this values down to around 2 cycles. However, even without additional hardware much better results can be achieved. The 31.4 % difference between the ECC programs show that there is still a lot of optimization potential on the compiler side as well. Implementations of cryptographic primitives should be protectable at hardly any cost given that their control flow is typically very sequential.

Enabling CSM on top of GPSA implies an additional overhead of up to 9 %. However, this is rather low considering that horizontal signatures with 4-bit (25 % redundancy per instruction) are used. Figure 4 shows how the runtime overhead of CSM scales in dependence of the horizontal signature size h. Most astonishing is probably that the overhead is still below 21 % even at 100 % redundancy (16-bit per instruction). The processor's Harvard architecture and the combination of 32-bit bus and 16-bit instruction set makes this possible.

6 Conclusion

This work extends the CFI concepts of Wilken and Shen from the soft error to the fault attack context. To achieve this goal we not only analyzed the functional requirements for derived signature calculation, but also performed an evaluation of actual signature functions. Using a CRC with suitable polynomial, any error in a single cycle and at least 7 bit-flips, spread across two cycles, can be detected with certainty.

We further practically implemented the derived signature based GPSA and CSM techniques for a state-of-the-art processor, the ARM Cortex-M3. Additionally, a toolchain for this platform has been created utilizing the LLVM compiler infrastructure. This toolchain incorporates all necessary transformations and is completely transparent for the programmer. As a result, arbitrary C programs can be protected by simple compilation. The design's low hardware overhead and the good detection capability indicates that the combination of GPSA and CSM is well suited to protect the control flow in the context of fault attacks.

7 Acknowledgements

This work has been supported by the Austrian Research Promotion Agency (FFG) under grant number 845579 (MEMSEC).

References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow Integrity Principles, Implementations, and Applications. ACM Trans. on Information and System Security (TISSEC) pp. 4:1–4:40 (Nov 2009)
2. Arora, D., Ravi, S., Raghunathan, A., Jha, N.: Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors. IEEE Trans. on VLSI Systems 14(12), 1295–1308 (Dec 2006)
3. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults. In: Advances in Cryptology, pp. 37–51. No. 1233 in LNCS, Springer (1997)
4. Free60.org: (2012), `http://free60.org/wiki/Reset_Glitch_Hack`
5. Joye, M., Tunstall, M. (eds.): Fault Analysis in Cryptography. No. 1619-7100 in Information Security and Cryptography, Springer (2012)
6. Lalande, J.F., Heydemann, K., Berthomé, P.: Software countermeasures for control flow integrity of smart card C codes. In: Computer Security-ESORICS 2014, pp. 200–218. Springer (2014)
7. Namjoo, M.: Techniques for Concurrent Testing of VLSI Processor Operation. In: International Test Conference. pp. 461–468. IEEE (Nov 1982)
8. Rodríguez, F., Campelo, J.C., Serrano, J.J.: A Watchdog Processor Architecture with Minimal Performance Overhead. In: Computer Safety, Reliability and Security, pp. 261–272. No. 2434 in LNCS, Springer (2002)
9. Rodríguez, F., Serrano, J.J.: Control Flow Error Checking with ISIS. In: Embedded Software and Systems, pp. 659–670. No. 3820 in LNCS, Springer (2005)
10. Wilken, K.D., Shen, J.P.: Continuous signature monitoring: efficient concurrent-detection of processor control errors. In: New Frontiers in Testing. pp. 914–925 (Sep 1988)
11. Wilken, K.D., Shen, J.P.: Continuous signature monitoring: low-cost concurrent detection of processor control errors. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 9(6), 629–641 (Jun 1990)
12. van Woudenberg, J.G.J., Witteman, M.F., Menarini, F.: Practical optical fault injection on secure microcontrollers. In: Fault Diagnosis and Tolerance in Cryptography (FDTC). pp. 91–99. IEEE (Sept 2011)

13. Zhang, M., Sekar, R.: Control Flow Integrity for COTS Binaries. In: Proceedings of the 22nd USENIX Conference on Security. pp. 337–352. SEC, USENIX Association, Berkeley, CA, USA (2013)