Securing Conditional Branches in the Presence of Fault Attacks

Robert Schilling*†, Mario Werner*, Stefan Mangard*
*Graz University of Technology
firstname.lastname@iaik.tugraz.at
†Know-Center GmbH

***Abstract*—In typical software, many comparisons and subsequent branch operations are highly critical in terms of security. Examples include password checks, signature checks, secure boot, and user privilege checks. For embedded devices, these security-critical branches are a preferred target of fault attacks as a single bit flip or skipping a single instruction can lead to complete access to a system. In the past, numerous redundancy schemes have been proposed in order to provide control-flow-integrity (CFI) and to enable error detection on processed data. However, current countermeasures for general purpose software do not provide protection mechanisms for conditional branches. Hence, critical branches are in practice often simply duplicated.**

We present a generic approach to protect conditional branches, which links an encoding-based comparison result with the redundancy of CFI protection mechanisms. The presented approach can be used for all types of data encodings and CFI mechanisms and maintains their error-detection capabilities throughout all steps of a conditional branch. We demonstrate our approach by realizing an encoded comparison based on AN-codes, which is a frequently used encoding scheme to detect errors on data during arithmetic operations. We extended the LLVM compiler so that standard code and conditional branches can be protected automatically and analyze its security. Our design shows that the overhead in terms of size and runtime is lower than state-of-the-art duplication schemes.

***Index Terms*—control-flow integrity, conditional branch, fault attacks, countermeasures**

I. Introduction

A conditional branch determines the program flow of the executed software based on a flag or based on the comparison of two values. While the basic functionality of a conditional branch is quite simple, the correct execution is highly critical for the security of computer systems. In the end, it is a conditional branch that decides whether or not an entered password is considered correct by a system, a system update is performed, a signature check is considered successful, or a user is granted access to a privileged function. The security implications are huge if a critical program flow decision is not taken correctly and for example unauthenticated software is executed [17].

Under normal conditions, conditional branches execute correctly, i.e., the branch is performed according to the comparison. However, there exist fault attacks, which allow an attacker to change the state of a system by using, i.e., a laser [21] or by manipulating the clock signal or the supply voltage [3]. The effects of fault induction can include the skipping of instructions [18], the redirection of memory accesses, or flipping or forcing bits in memory or registers. Even bypassing secure boot mechanisms is possible [17].

The research field of studying fault inductions on the security of cryptographic algorithms is very active [9], [20]. Techniques exist to reveal the keys of different cryptographic functions for many different fault models [6], [7]. There also exist several proposals for countermeasures [14], [15]. When it comes to attacks and countermeasures for the general execution of software, there are significantly fewer publications. Countermeasures that have been proposed so far for securing software execution can be grouped into two classes. First, there are countermeasures that aim for ensuring control-flow integrity (CFI) in the setting of fault attacks, like [23]. Concerning conditional branches, these countermeasures only ensure that one of the two possible execution paths and no completely different path is taken after a conditional branch. However, these CFI countermeasures do not protect the decision which path is taken against fault attacks. The second class of countermeasures are redundancy mechanisms for data [12], [22]. For example, [11] shows how to protect variables during arithmetic operations using AN-codes. However, as also pointed out in [13], such schemes only protect data values and their processing and no branching operations based on the data.

Today, there is a gap. There exist CFI and data protection schemes against fault attacks. However, there is no method of linking them efficiently such that the decision on which execution path to take is protected at the same level as the control-flow and the processing of the data. In practice, a method to avoid this gap is to do not only one conditional branch, but to check the condition for the branch again after the branch has been taken. This duplication approach increases security and can be scaled to an arbitrary order. However, this duplication approach leads to significant overheads on the one side, and it can be attacked by inducing multiple times the same fault. The options for creating diversity by using different branches to make attacks harder is limited. Typically, it is the same hardware multiplexer for all branches, which decides which address is loaded next and remains as a single point of failure. In this article, we close the existing gap by providing protection for conditional branches which is encoding-based, like the redundancy schemes for data and CFI. Our concrete contributions are as follows:

1) We present a generic solution that closes the gap of unprotected conditional branches in the presence of a CFI protection scheme. Conditional branches are protected by linking a redundant comparison result with the redundancy of the CFI protection scheme.
2) We show that we can use AN-codes to efficiently perform a redundant comparison of encoded values which preserves the redundancy.
3) We present an LLVM compiler extension to automatically identify and protect conditional branches based on the concept of AN-codes. We provide experimental results showing that the overhead in terms of code size and runtime is lower than state-of-the-art duplication schemes. Furthermore, a bootloader application can efficiently be protected with 2.4% code overhead and with less than 0.1% runtime overhead.

The structure of this paper is as follows. We define the threat model and discuss existing countermeasures against faults in Section II. In Section III, we present how we protect conditional branches in this setting. We discuss a novel approach to compute a redundant comparison result used for protected conditional branches in Section IV. In Section V, we present a compiler extension to protect conditional branches automatically and evaluate the overhead. Section VI analyzes the security of the countermeasure, and finally, in Section VIII we conclude this paper.

II. Fault Protected Software Execution

Throughout this paper, we consider an attacker with physical access to an embedded system. Hence, the attacker can tamper with the surroundings to induce faults, which can modify data, code, and also signals like the comparison result. Faults can occur once or multiple times with multiple bits modified. We assume the presence of an instruction-granular CFI protection scheme, protecting the execution of instructions and the selection of the operands. Furthermore, we assume the data to be encoded redundantly.

A. Code Protection with Control-Flow Integrity

CFI protection schemes [1] typically protect the execution of code against software attacks. However, recent work extends these countermeasures in the context of fault attacks, where they try to protect the sequence of basic blocks or even the sequence of instructions [10], [19], [23].

CFI protection schemes in the context of fault attacks rely on an internal state S, which is modified by each executed instruction. Independent of the concrete CFI protection scheme, control-flow transfers like conditional branches require special treatment. On control-flow transfers, the internal state S diverges, because the instructions diverge. When the control-flow graph merges at a later point in the program, the CFI state S also needs to merge. To support this, CFI protection schemes either use correction values or replace the state.

Although CFI protection mechanisms can deal with conditional branches, they can not protect them. Such a scheme only ensures that one of the correct successor' s blocks is executed after the branch, but the correct selection is completely unprotected leaving a single point of failure.

B. Data Protection with AN-Codes

To counteract soft errors in modern CMOS devices caused by radiation, encoding schemes like [12], [22] have been proposed. However, these mechanisms are developed for a protected storage in the memory not for protecting operations inside the CPU. We focus on AN-codes [8], which are well suited for fault protection [11], and natively supports different arithmetic operations. AN-codes have the form $n_c = A \cdot n$, where n_c denotes the code word, A the encoding constant, and n the functional value. Hence, all multiples of A are valid code words. To validate the code word, the AN-code congruence in the form $0 \equiv n_c \mod A$ is applied.

AN-codes limit the functional value to be less than A to preserve the error detection capabilities. The encoding constant is chosen by the designer and defines the redundancy properties of the code. The minimum Hamming distance between the code words gives a quantitative measure how strong the chosen A is. Finding a good A so far is limited by exhaustive search [16], but good encoding constants already have been found. Hoffmann et al. [13] call these constants as so-called *Super A*s because their minimum Hamming distance is maximal for a given word width.

Since AN-codes are closed under addition and subtraction (Equation 1), these operations do not require any modification. Operations like multiplication are supported but require a special correction value.

$$z_c = x_c + y_c = A \cdot x + A \cdot y = A \cdot (x + y) = A \cdot z \qquad (1)$$

Fetzer et al. [11] use this encoding scheme to build an AN-code LLVM compiler, which transforms all operations to the domain of AN-codes, to protect the data processing. However, as discussed by Hoffmann et al. [13], AN-codes alone is not sufficient because conditional branches are still a single point of failure.

C. Duplication

One way to avoid different schemes for protecting code execution and data is modular redundancy [4], where each instruction is duplicated. After a duplicated instruction, a check is inserted. Conditional branches are protected by replicating the branch multiple times resulting in a comparison tree. However, inducing the same fault multiple times bypasses this protection. Barry et al. [5] automate this, where they duplicate instructions and store the result always in the same result register and avoid check operations. However, this countermeasure is only suitable in the instruction skip fault model.

III. Protecting Conditional Branches

A conditional branch consists of two operations: a comparison and a branch. The comparison takes two inputs x and y, compares them with a predicate P (e.g., $<$), and results in a 1-bit signal indicating if the comparison is true

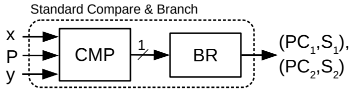

Figure 1. Conditional branch with CFI state.

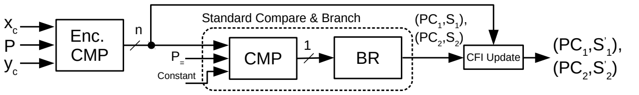

Figure 2. Protected conditional branch with state update and n-bit redundantly encoded comparison.

or false. Typically, this signal is part of the CPU flags. The branch takes this signal and decides how to update the program counter (PC), which can end up with two different values PC_1 and PC_2, depending on whether the branch was taken or not.

In the presence of a CFI protection mechanism, conditional branches work differently. Again, there is a compare and branch operation as shown in Figure 1. However, the CFI protection mechanism contains a dedicated internal state S for each value of the PC, which is updated when executing the conditional branch. Here, the output of a conditional branch is two different PC values PC_1 and PC_2 with their corresponding CFI states S_1 and S_2.

However, even in the presence of a CFI protection scheme, there are three different error sources, which are not protected and can lead to a wrong execution:

1) *Faulting the operands.* Modifications on the branch operands or any data that leads to the comparison can result in a wrongly executed conditional branch.
2) *Faulting the comparison.* The value deciding whether a conditional branch is taken or not, the condition signal, is a 1-bit signal. An attacker being able to control this signal precisely can change the execution of the conditional branch.
3) *Faulting the branch.* A fault modifies the execution of the branch such that the branch is taken although the condition value says otherwise or vice versa.

To protect conditional branches, we assume that data and all performed operations on it are encoded redundantly, e.g., via AN-codes. We generically address the latter two points as follows: first, we use a redundantly encoded condition computation, to ensure the integrity of the condition value. This encoded comparison takes two encoded values x_c and y_c, a comparison predicate P, and outputs a redundantly encoded condition result, which Hamming distance is large enough to maintain the same security level throughout the whole conditional branch. The comparison predicate P does not require redundancy by means of encoding since a different predicate uses a different expected condition value.We then use the standard compare and branch mechanism that compares the redundant comparison result with one of the expected condition values.

Without further measure, this introduces an intermediate unprotected 1-bit signal. To mitigate this, we exploit the redundancy of the encoded comparison result and merge this value as part of the CFI state update into the redundancy of the CFI scheme (Figure 2). Only if the condition is computed correctly and the branch was executed correctly, the states for S_1' and S_2' are correct. This approach eliminates the single point of failure present in state-of-the-art CFI protection schemes by not relying on a 1-bit condition value but on a redundantly encoded condition value linked with the CFI state. The comparison is protected by using an encoded comparison operation that yields a redundant result. The final conditional branch is protected by linking the redundant condition value with the CFI redundancy. Fault attacks in both cases yield an invalid state S, which is detectable.

Using an encoded comparison operation ahead of an ordinary conditional branch makes this design modular and flexible allowing different encodings with different security levels to be used at various program locations. The only requirement for the CFI protection is the ability to merge a value into the internal state. A dedicated conditional branch, which automatically performs the CFI state update and linking the result of the encoded condition value with the CFI redundancy can increase the performance.

IV. Protected Comparisons with AN-Codes

In this section, we discuss a redundant comparison framework which is exploiting the arithmetic properties of AN-codes, which adheres to the interface definition in Equation 2. The inputs, the internals, and the output are encoded such that there is no single point of failure. The two possible outputs of the encoded comparison operation should have a Hamming distance larger or equal than a constant D, where D denotes the minimum security level in bits of the data encoding and the CFI redundancy. Furthermore, we want to avoid the all-zero, and all-one condition results because faulting to these values is easier than to others due to the hardware implementation (e.g., the reset line of a register can initialize its value to zero).

$$\begin{aligned} \text{condition} \leftarrow & \text{EncodedCompare}\,(P, x_c, y_c) \\ & \text{with condition} \in \{C_1, C_2\} \text{ and} \\ & \text{Hamming distance}\,(C_1, C_2) \geq D \end{aligned} \tag{2}$$

AN-codes can be compared using a standard compare instruction. However, this removes all redundancy and results in a 1-bit signal stored inside the CPU. Hoffmann et al. [13] found this issue during fault simulation. Instead, we compute the comparison and preserve the redundancy of the AN-codes avoiding this single point of failure.

To compute the $x_c < y_c$ comparison (x_c and y_c are AN-coded), we start with a subtraction. Based on the sign of this result, we get the information shown in 3. However, we cannot directly use the sign bit because it is not redundant. The challenging task is performing an entropy compression, where we map the encoded positive difference values to C_1, and all encoded negative values to C_2. Additionally, we want to maximize the Hamming distance between C_1 and C_2 yielding a redundant comparison result.

Algorithm 1: AN-encoded < comparison.

```
Data: $x_c, y_c \in$ AN-code, $0 < C < A$.
Result: $cond \in \{C_1, C_2\}$.
begin
    diff ⟵ (unsigned) $x_c - y_c + C$
    cond ⟵ diff % A
end
```

$$x_c - y_c \begin{cases} \text{positive if} & x_c \geq y_c \\ \text{negative if} & x_c < y_c \end{cases} \tag{3}$$

Our approach *arithmetically* computes this entropy compression yielding a comparison result which preserves the redundancy of the AN-code. When looking at the difference in Equation 3, the congruence $0 \equiv (x_c - y_c) \mod A$ is valid because AN-codes are closed under subtraction in a signed representation. However, when interpreting the AN-code congruence in an unsigned representation, this destroys the congruence for negative differences. For a positive difference, on the other hand, the unsigned representation does not change anything. By intentionally destroying the AN-code congruence for negative numbers due to casting to unsigned, we are able to separate the two cases of Equation 3 yielding two different values. Using 32-bit data types, the unsigned interpretation x_u of a signed negative value $x_s < 0$ in the twos-complement representation is computed as $x_u = 2^{32} + x_s$. We exploit this property of twos-complement encoded negative numbers for the required entropy compression. First, the difference is cast to an unsigned value. This does not change the difference if it was positive. Negative values change according to the twos-complement, where the AN-encoded difference becomes invalid. In Equation 4, we show the conversion from the signed AN-code to the unsigned representation for negative values of the difference.

$$(x_c - y_c)_u = 2^{32} + (x_c - y_c) = 2^{32} + A \cdot (x - y) \tag{4}$$

When applying the AN-code congruence to that value by using a modulo operation with A, we obtain a dedicated value for the negative difference as shown in Equation 5.

$$\left(2^{32} + A \cdot (x - y)\right) \% A = 2^{32} \% A \tag{5}$$

The relation described before only holds true for the negative difference. For a positive difference in Equation 3, the AN-code congruence still returns zero. However, as discussed before, having a comparison result that is zero is not favorable. We avoid this zero comparison result for the true case by adding constant $0 < C < A$ to the difference before we compute the remainder (this also changes the comparison result for the false case).

Algorithm 1 summarizes how the encoded less-than comparison is computed. The comparison result *cond* holds the value $2^{32}\%A + C$ if x_c is less than y_c or the value C if x_c is larger or equal than y_c. A modification (e.g., due to a fault) to the operands such that their AN-code gets invalid results in a different comparison result, making it invalid.

Algorithm 2: AN-encoded = and ≠ comparison.

```
Data: $x_c, y_c \in$ AN-code, $0 < C < A$.
Result: $cond \in \{C_1, C_2\}$.
begin
    diff1 ⟵ (unsigned) $x_c - y_c$
    diff1 ⟵ diff1 + C
    rem1 ⟵ diff1 % A
    diff2 ⟵ (unsigned) $y_c - x_c$
    diff2 ⟵ diff2 + C
    rem2 ⟵ diff2 % A
    cond ⟵ rem1 + rem2
end
```

The same scheme applies to compute a ≤, >, and ≥ comparison by swapping the operands in the first subtraction and swapping the symbols for the true and false case, as summarized for 32-bit data types in Table I.

Protected Equal and Not-Equal Condition Computation

To compute the = and ≠ condition, we combine the ≤ and ≥ condition. The = condition is true if both conditions are true and false if only ≤ is true or ≥ is true. Both conditions cannot be false at the same time. We combine these conditions using an addition. Using the condition values for ≥ and ≤ from Table I, the sum of both true values is $2 \cdot C$. The false case is the sum of one true and one false case resulting in the condition value $2^{32}\%A + 2 \cdot C$. The algorithm to compute the = or ≠ condition is shown in Algorithm 2.

a) Parameter Selection.: For the comparison algorithms, we used 32-bit registers and chose A to be 63877 (a *super*-A according to Hoffmann et al. [13]). This A maximizes the functional value for 16-bit data and has a minimum Hamming distance of *six* between all code words, allowing the code to detect up to 5-bit errors. We then chose C such that it maximizes the Hamming distance between the true and false symbol for one comparison. For the = and ≠ comparison we select $C = 14991$ and for the $<, \leq, >, \geq$ comparison we select $C = 29982$. With both constants, we reach a maximum Hamming distance D of 15-bit between the comparison values.

V. Implementation and Evaluation

We included all transformations to the LLVM compiler (Figure 3) and evaluated this scheme using an ARMv7-M instruction set architecture (ISA) simulator. We use a software-centered GPSA CFI scheme similar to the one in [23]. The branch protection is purely implemented in software and does not require hardware modifications.

The compiler front end contains a new function attribute (i.e., `protect_branches`) to annotate functions that require

Table I
Condition values for encoded $<, \leq, >, \geq$ condition values.

Predicate	Subtraction	True Value	False Value
>	$y_c - x_c$	$2^{32}\%A + C$	C
≥	$x_c - y_c$	C	$2^{32}\%A + C$
<	$x_c - y_c$	$2^{32}\%A + C$	C
≤	$y_c - x_c$	C	$2^{32}\%A + C$

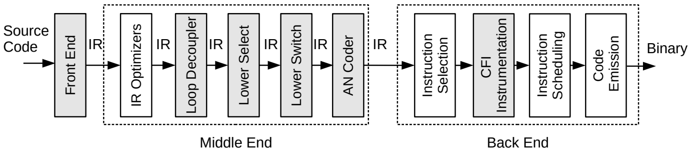

Figure 3. Modified LLVM compiler pipeline. Grey boxes indicate modifications or additions of/to the regular compilation flow.

protection. The AN-code instrumentation is performed in the middle end. There, the optimized intermediate representation (IR) is preprocessed by a custom *Loop Decoupler* pass which separates loop induction variables from the use in arithmetic expressions or memory accesses and a *Lower Select/Switch* pass simplifying the IR for the subsequent *AN Coder*. The *AN Coder* pass transforms all instructions, which end up in the comparison operation of a conditional branch to the AN-domain. Moreover, the AN-code based *encoded compare* is added here. Up to this point in the compiler pipeline, all transformations are independent of the target architecture and CFI scheme. The *CFI Instrumentation* pass in the back end is the only architecture and CFI specific part of this design. It performs the CFI instrumentation and adds the *state updates* to the conditional branches.

a) Cost Analysis.: The overhead of our implementation comprises three parts: the cost of computation on encoded data yielding into a branch, the costs of the branch protection scheme, and the costs of the CFI scheme. Given that we solely propose a branch protection, we do not focus on analyzing the cost of the used data protection or CFI scheme. These costs are highly application specific and therefore hard to predict. Still, our evaluation indicates that the expected costs for enforcing CFI and for protecting data values are quite reasonable when mostly requiring arithmetic operations.

Analyzing the cost of the encoded compare and the state update operations (Table II) is possible precisely. The generic implementation[1] of the proposed encoded compare comprises additions, subtractions, and modulo operations. Every ISA typically supports addition and subtraction, but modulo is not necessarily supported directly and therefore often is more costly. With the used ARMv7-M ISA, modulo has to be implemented using a combination of a slow division (UDIV) and a multiply+subtract (MLS) instruction. As a result, depending on comparison predicate, between 12 and 26 bytes memory overhead, and 6-33 cycles runtime overhead is generated for one encoded compare. Hardware support for a fast modulo instruction would considerably reduce this overhead.

Table II
Qualitative overhead analysis of the building blocks.

Predicate	Required Operations	Our Prototype		
		Instructions	Size / B	Runtime / c[a]
$>$ $\geq$ $<$ $\leq$	1 + 1 - 1 %	1 ADD 1 SUB 1 UDIV 1 MLS	12	6-16
$=$ $\neq$	3 + 2 - 2 %	3 ADD 2 SUB 2 UDIV 2 MLS	26	13-33

[a] Division on ARMv7-M requires between 2 and 12 cycles.

Table III
Size and runtime overhead of different branch protections.

Benchmark	Metric	CFI	Duplication		Prototype	
		abs	abs	+ / %	abs	+ / %
integer compare	Size / B	12	128	967	86	617
	Runtime / c	20	91	355	63	215
memcmp	Size / B	68	272	300	276	306
	Runtime / c	1689	10210	504	8905	427
bootloader	Size / B	17252	—	—	17672	2.435
	Runtime / c	51888k	—	—	51888k	0.001

The cost for state updates dependents on the CFI scheme. In the software-centered design, they are implemented using one address load and a store of the comparison result to the CFI unit. These instructions are added to the beginning of the successor basic blocks of the protected conditional branch and introduce 4 bytes code and 4 cycles of runtime overhead per instantiation. An optimized CFI and branch protection design can fully omit these costs.

b) Benchmarks.: We use two micro-benchmarks to measure the overhead in terms of runtime and code size. These benchmarks (*integer compare* and *memcmp*) test the branch protection in isolation by exercising a single integer equal comparison and a secure memory comparison with 128 elements. We compare this overhead with a duplication approach, where we duplicate the conditional branch six times consecutively to have a comparable single bit fault tolerance to the AN-code based implementation (i.e., 6-bit Hamming distance for the encoded values). However, this duplication approach does not protect any data or arithmetic operation leading to the branch opposed to the AN-code based scheme. As a macro-benchmark, we implement a fault-protected version secure bootloader, similar to the one in [2]. Only programs which feature a valid ECDSA signature over the program's hash get executed. In this example, the memory comparison of the signature verification and all subsequent conditional branches are protected. This mitigates the single point of failure of a secure boot mechanism, which was already a target of fault attacks.

The costs (Table III) also include the overhead of computing on the AN-encoded values. Based on the micro-benchmark results, we observe that the performance in terms of code size and runtime is on par with the duplication approach or even better. However, we do not only protect the conditional branch but also protect the data and the arithmetic operations on it. When applying this protection mechanism to the protected bootloader, the overhead is neglectable since the crypto implementation dominates code size and runtime. The code size overhead of less than 2.5% and a neglectable runtime overhead makes this countermeasure applicable to real-world applications.

[1] Special encoding constants may have optimized implementations but different code properties.

VI. Security Analysis

To state the security of the countermeasure, we analyze its fault resistance. If there is a fault on a single location but with multiple bits flipped, the error is transparent and detectable relying on the code properties of the selected A [11]. For our parameter selection, we can detect up to 5-bit errors in a single word during the calculation. In the final condition result, the error detectability is even higher because only two symbols are valid. At this place, we reach a Hamming distance of 15-bit between the two condition values.

However, if errors are spread over multiple locations/operations, the fault detection capabilities of the AN-code decrease and the code cannot detect as many bits as before. To investigate this behavior, we performed a simulation with faults at different locations. Simulations show that for our parameter selection the error detectability is reduced to 3-bits, arbitrarily placed over all the whole computation of the condition value. With four bits flipped over the whole computation of a condition value, the error rate where an attacker can flip the final condition value from true to false or vice versa is 0.0002 %, which increases having more bits flipped.

VII. Acknowledgement

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402) and by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria.

VIII. Conclusion

In this work, we close the gap of unprotected conditional branches in CFI countermeasures in the presence of fault attacks. We eliminate the single point of failure by adding an encoded comparison operation that yields a redundant condition value. Using a standard compare and branch mechanism together with the ability to merge the redundant comparison result with the CFI protection mechanism allows us to protect the execution of a conditional branch. Our approach is highly flexible allowing us to use different encoded comparison operations based on different encoding schemes with different security properties at different places in the program. We exploit the properties of arithmetic AN-codes and present novel comparison algorithms to compute the condition values arithmetically but preserve the redundancy. We integrated this countermeasure in the LLVM compiler to automatically protect conditional branches. Experimental evaluation shows little overhead to security critical programs such as the signature verification of a secure bootloader making it applicable for real-world usage.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Conference on Computer and Communications Security – CCS 2005.*

[2] Atmel, "Atmel at02333: Safe and secure bootloader implementation for sam3/4," http://www.atmel.com/Images/Atmel-42141-SAM-AT02333-Safe-and-Secure-Bootloader-Implementation-for-SAM3-4_Application-Note.pdf, [accessed 19-July-2017].

[3] A. Barenghi, G. Bertoni, E. Parrinello, and G. Pelosi, "Low voltage fault attacks on the RSA cryptosystem," in *Fault Diagnosis and Tolerance in Cryptography – FDTC 2009.*

[4] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Low Cost Software Countermeasures Against Fault Attacks: Implementation and Performances Trade Offs," *Proceedings of 5th Workshop on Embedded Systems Security - WESS*, 2010.

[5] T. Barry, D. Couroussé, and B. Robisson, "Compilation of a countermeasure against instruction-skip fault attacks," in *Cryptography and Security in Computing Systems – CS2@HiPEAC 2016.*

[6] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology – CRYPTO 1997.*

[7] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of eliminating errors in cryptographic computations," *J. Cryptology*, 2001.

[8] D. T. Brown, "Error detecting and correcting binary codes for arithmetic operations," *IRE Trans. Electronic Computers*, 1960.

[9] C. Chen and S. Yen, "Differential fault analysis on AES key schedule and some coutnermeasures," in *Information Security and Privacy – ACISP 2003.*

[10] R. de Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. D. Bosschere, B. Preneel, B. D. Sutter, and I. Verbauwhede, "SOFIA: software and control flow integrity architecture," in *Design, Automation & Test in Europe Conference & Exhibition – DATE 2016.*

[11] C. Fetzer, U. Schiffel, and M. Süßkraut, "An-encoding compiler: Building safety-critical systems with commodity hardware," in *Computer Safety, Reliability and Security – SAFECOMP 2009.*

[12] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, 1950.

[13] M. Hoffmann, P. Ulbrich, C. Dietrich, H. Schirmeier, D. Lohmann, and W. Schröder-Preikschat, "A practitioner's guide to software-based soft-error mitigation using an-codes," in *IEEE International Symposium on High-Assurance Systems Engineering – HASE 2014.*

[14] C. H. Kim and J. Quisquater, "Fault attacks for CRT based RSA: new attacks, new results, and new countermeasures," in *Information Security Theory and Practice – WISTP 2007.*

[15] T. Malkin, F. Standaert, and M. Yung, "A comparative cost/security analysis of fault attack countermeasures," in *Fault Diagnosis and Tolerance in Cryptography – FDTC 2006.*

[16] M. Medwed and J. Schmidt, "Coding schemes for arithmetic and logic operations - how robust are they?" in *Information Security Applications – WISA 2009.*

[17] Riscure, "Bypassing Secure Boot using Fault Injection," https://www.blackhat.com/docs/eu-16/materials/eu-16-Timmers-Bypassing-Secure-Boot-Using-Fault-Injection.pdf, [accessed 13-September-2017].

[18] J. Schmidt and C. Herbst, "A practical fault attack on square and multiply," in *Fault Diagnosis and Tolerance in Cryptography – FDTC 2008.*

[19] D. Sullivan, O. Arias, D. Gens, L. Davi, A. Sadeghi, and Y. Jin, "Execution integrity with in-place encryption," *CoRR*, 2017.

[20] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault," in *Information Security Theory and Practice – WISTP 2011.*

[21] J. G. J. van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *Fault Diagnosis and Tolerance in Cryptography – FDTC 2011.*

[22] Z. Wang, M. G. Karpovsky, and K. J. Kulikowski, "Replacing linear hamming codes by robust nonlinear codes results in a reliability improvement of memories," in *Dependable Systems and Networks – DSN 2009.*

[23] M. Werner, E. Wenger, and S. Mangard, "Protecting the control flow of embedded processors against fault attacks," in *Smart Card Research and Advanced Applications – CARDIS 2015.*