Nonlinear Code-based Low-Overhead Fine-Grained Control Flow Checking

G. Dar, Giorgio Di Natale, O. Keren

Nonlinear Code-based Low-Overhead Fine-Grained Control Flow Checking

Gilad Dar[‡], Giorgio Di Natale[†], Osnat Keren[‡]
[†]Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA, 38000 Grenoble, France
[‡]Faculty of Engineering, Bar-Ilan University

***Abstract*—A hardware-based control flow monitoring technique enables the detection of errors in both the control flow and the instruction stream executed on a processor. However, as shown in recent publications, these techniques fail to detect malicious carefully-tuned manipulations of the instruction stream in a basic block. This paper presents a non-linear encoder and checker that can cope with this weakness. It is a MAC based control flow checker that has the advantage of working with basic blocks of variable length, can detect every error, and performs the computation in real-time. The architecture can easily be modified to support different signature size and error masking probabilities.**

***Keywords*—** Embedded Security, Control Flow Checking, Non-Linear Codes, Signature, Countermeasures

I. Introduction

Dependability is an important characteristic of modern computing systems. The hardware components of a system can be affected by faults deriving from different root causes such as environmental perturbations (e.g., radiation, electromagnetic interference) or malicious attacks (e.g., fault attacks, software modification or replacement).

Many techniques have been proposed to cope with transient, permanent and malicious fault. These techniques for reliability improvement and fault tolerance target both the hardware and the software, and rely on different forms of redundancy. Among them, Control Flow Checking (CFC) makes it possible to cover faults affecting storing elements containing the executable program, as well as all the hardware components handling the program itself and its flow [1, 2]. It can also cope with the effects of a malicious attacker who tries either to bypass security checks or retrieves secret information by fault injection [3, 4].

Software based CFC solutions that modify the program rely on the assumption that the binary code stored in memory is not being maliciously tampered. Thus, these solutions cannot provide security against fault injection attacks [5]. In contrast, hardware-based CFC solutions, such as [6] can detect malicious code and data tampering at run-time.

There are two types of hardware-based CFC policies: fine grained and coarse grained [5]. A fine grained CFC policy allows control flow along the valid edges of the Control Flow Graph (CFG), whereas a coarse grain policy relaxes this restriction. A CFG makes it possible to model the normal program behavior of a code that is not self-modifying or generated on the fly as a walk on a static graph. The nodes in this graph are sequences of non-branching instructions (also called basic blocks) with a single entry point at the first instruction and a single exit point at the last instruction. The edges of the graph represent jumps, branches and returns. In [7] the authors distinguished between two levels of fine granularity: *instruction integrity checking* which aims to detect attacks which may not result in control flow violations, and *instruction flow checking* for detecting forward-edge and backward-edge flow violations between basic blocks.

In [8] the authors suggested encrypting the instructions to detect changes in it. The instructions are decrypted by adding a stage to the pipeline, immediately before the instruction's decode stage, which required architectural modifications. The additional stage of the pipeline introduced an $\approx 9.1\%$ total execution time overhead. At this point, it is important to note that tampering the *flow* of the program (i.e., its branches, jumps, calls and returns) can affect its behaviour tremendously. For that reason, the authenticity of the last instruction in a basic block must be verified as close as possible to its execution time, as the *Signature Modeling* approach suggests.

Signature Modeling is a fine-grained technique [9, 10, 11]. In Signature Modeling, basic blocks are accompanied by a signature, such as a Cyclic Redundancy Check (CRC) checksum or Hamming code, that are generated at run-time and then compared against a pre-computed signature which is stored in a tamper-resistant memory (e.g., the tamper-resistant RAM presented in [12]). In the case of modification of any bit belonging to that portion of the code, the detection code deviates from the expected signature and reveals the fault. The two signatures can be compared during the execution of each instruction [10, 11] or when a basic block ends [7, 3]. In [11] a *CRC-based* signature monitor was integrated into the instruction fetch state to prevent the processing of instructions whose pre-calculated and current signatures do not match. However, a *CRC-based* signature monitor has a major drawback, it can be bypassed by a sophisticated attacker [13]. The authors in [14] proposed a technique to map one malicious software into another (protected by a control flow checking mechanism), without violating the structure of the latter; i.e., without being detected by a control flow monitoring technique. The basic principle involved the fine-tuning of the instructions in each basic block so that the generated signature corresponded to the one for the original program. In this paper we close this gap, we propose a fine-grained MAC-based CFC which utilizes *non-linear codes* to protect against malicious modifications of the executed program. We assume the attacker knows the protected architecture details and its machine language, as well as the program and its control flow graph. The attacker can execute malicious physical manipulations on the device by injecting precise faults at run-time into the machine code stored in memory. We assume that the signature is stored in a tamper-resistant memory that cannot be tampered with.

The contribution of this paper is:

- A non-linear code based on a weakened version of the Karpovsky-Wang Algebraic Manipulation Detection (AMD) code with multiple random variables [15].
- A signature calculation method that works in parallel to the processor pipeline and does not require processor changes, nor code changes.
- We introduce an upper bound on the probability that an error will not be detected. This bound applies to every basic block and hence obviates the need for simulations/experiments.
- The area overhead of the signature calculation is relatively small (compared to methods preventing malicious attacks) and there is no need to partition the program into basic blocks of equal length as required in [7].

This paper is organized as follows: Section II presents an overview

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant No. 1266/20) and by the IRS project CROCHET funded by the IDEX UGA

[*]Institute of Engineering Univ. Grenoble Alpes

of existing CFC solutions, and details the architecture in which the proposed signature calculation can be used. Section III provides formal definitions and formulates the security metric we use to evaluate the effectiveness of the construction. Section IV presents the theoretical construction of the non-linear code and Section V describes its hardware implementation. Finally, we draw some conclusions in section VI.

II. Context

Historically, error-detection codes primarily targeted natural faults which are likely to cause a small number of bit-flips, randomly distributed. These codes are usually linear and hence have a small overall impact on the target system in terms of area overhead and the additional delays introduced for their calculation. However, they cannot cope with malicious attacks. For example, [11], the so-called "derived signature" enables a checksum computation with zero latency. However, it utilizes systematic encoders of *linear cyclic codes* defined by generator polynomials over a finite field. Due to the linearity of the codes, the corresponding CFCs can only detect a (relatively) small number of errors, and they cannot detect attacks launched by sophisticated precise attackers.

Malicious attacks are handled better by non-linear methods; e.g., methods based on Message Authentication Codes (MAC). In MAC, the signature is calculated by resorting to secret information which also guarantees the authenticity of the data. MAC techniques that are based on a statically computed cryptographic hash of the instruction sequence in the basic block [16] generally have high latency, because the monitor has to buffer the instruction stream corresponding to a basic block and only start to compute the hash when the block ends. In some hash algorithms the input is processed through several rounds and additional latency is accumulated. A few MAC based checkers ([6]) can compute the signature together with the execution of the program itself. Nevertheless, these solutions have certain limitations. In [6] a Cipher Block Chaining-Message Authentication Code (CBC-MAC) algorithm with a 64-bit MAC length is used. Since CBC-MAC is only secure for messages of a fixed length, two block lengths of 5 and 6 instructions are supported only. In addition, its implementation has a critical path which is longer than the one of the processor, leading to a cycle overhead of 13.7% and a total execution time overhead of 110%. In [17] the authors use public-key cryptography to protect their code. The strength of the used cipher guarantees the security of the solution, however its cost (in hardware and timing overhead) is extremely high. In [18] the authors implemented *CCFI-cache*; a dedicated tamper-resistant signature memory with the same properties as the instruction cache. Each signature consists of the hash value of the instructions and the meta-data of the basic block along with the meta-data itself (The number of instructions in the basic block and the address of the next basic blocks).Since the signature may occupied several entries in the CCFI-cache, in the case where the basic block is very short, it must be padded with `nop` instructions, so its size match the size of the signature. Similarly, short signatures must be padded with empty entries in the case of a long basic block. This lead to a program overhead of up to 30%.

Here, we introduce a MAC scheme that can be applied to every architecture where the CFC (or *watchdog*) is a standalone module that works in parallel with the main processor's pipeline (see the generic architecture in Fig. 1). The CFC is a co-processor that calculates the signatures of the basic blocks by fetching the instructions to be executed from the main bus, and then comparing the obtained signature with a predefined one. It does not modify the pipeline stages, does not add latency, and does not interfere with the program flow. The CFC communicates with the processor via the existing signals at its interface or within the pipeline. Namely, the current address and instruction on buses used by the processor during the Instruction Fetch (IF) phase, and the calculated address of the next instruction. For instance, the proposed approach can be easily integrated in recent solutions, such as [19], where the checker is an independent module as the one shown in Fig. 2. The generic architecture we consider consists of four main blocks: a compact processing unit, a comparator,

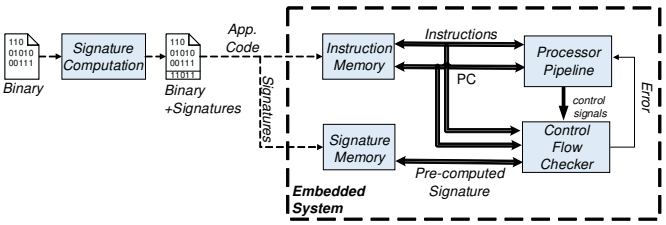

Figure 1: Generic architecture for fine-grained signature based control flow checkers.

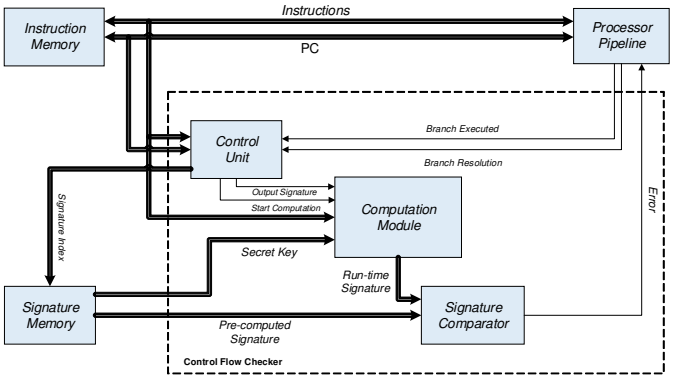

Figure 2: Micro-architecture of a fine-grained control flow checker.

a control unit, and a tamper-resistant memory array. The tamper-resistant memory is more expensive, but is only used for storing a small amount of information, not the whole program. The size of the tamper-resistant memory and its width depends on the number of basic blocks, their maximal size and the required security level.

In this paper we focus on the design of the *computation module*. We assume that:

- The control unit generates all the control signals for the computation module. This includes the generation of a reset signal that goes to the computation module at the beginning of a basic block and an indication that the basic block ends (due to a branch instruction or because it has reached its maximal size).
- The control unit delivers the signature from the tamper-resistant memory to the computation module.

We also assume that:

- The profiling process as well as the program can be trusted.
- The content of the tamper-resistant memory is pre-computed from the control flow graph (see the process flow diagram in Fig. 1). The pre-computation of the signatures (i.e., the encoding of the basic blocks by using random vectors) can be trusted.
- The process of loading the signatures into the tamper-resistant memory can be trusted.
- To reduce the cost of the product, the main memory has no dedicated security protection whereas the CFC itself, including its tamper-resistant memory in which the signatures are stored, is not accessible to the attacker.
- The attacker knows the original code, its profiling and its location in the main memory.
- The attacker is able to tamper with the content of the main memory and is able to inject arbitrary or precise errors before the execution or at run-time; i.e., when the code is being fetched from the memory.

III. Definitions and Security metric

A. Basic blocks

A basic block is a piece of code made of one or several consecutive instructions without any jumps between them. A basic block starts when its address is the target of a jump instruction of another basic block, and ends with a jump to another basic block (or a return), or if the successor instruction is a target of a jump instruction. For example,

Example 1. *The following code consists of 3 basic blocks:*

```
    ...
    DEC R1
    BNZ R1, L1          //end of BB1
    INC R1              //end of BB2
L1: ADD R2, 2           //beginning of BB3
    ...
```

The first basic block ends because of jump instruction and the second due to the fact that 'ADD R2, 2' is a target of a jump instruction. □

The size of the BB is defined as the number of bytes occupied by all the instructions in that basic block. Its size can range from 32 (for a 32-bit architecture) to $8N$ bits (N bytes). If the original basic block is larger than N bytes, it should be divided into several basic blocks by inserting a branch instruction which jumps to the successor instruction.

The content of a basic block can be referred to as a binary string, or, as is common in coding theory, as a q-ary vector over an alphabet of size $q = 2^r$. When algebraic codes are used for signature computation, the symbols of the vectors are treated as elements in the finite field $\mathbb{F}_q = GF(q)$. The size of the alphabet determines both the effectiveness of the code and the implementation complexity. In general, a small r lowers the implementation cost of the multipliers over that field, whereas a large r increases the fault detection probability.

B. Signature structure

The signature S is a q-ary vector of length $t+1$ symbols (i.e., $(t+1)\cdot r$ bits). It has two parts: a "key" X and a tag f; that is, $S = (X, f(X,Y))$. In this paper, the "key" is a non-zero vector, $X = (x_t, ...x_1) \in \mathcal{X} \subseteq \mathbb{F}_q^t$ chosen at random by the manufacturer. Y is the content of a basic block

$$Y = (y_k, ...y_1) \in \mathbb{F}_q^k,$$

and $f = f(X,Y) \in \mathbb{F}_q$ is a **single** q-ary symbol that represents the tag. The effectiveness of the CFC is determined by the choice of the tag function.

The triplets $(Y, X, f(X,Y))$ form a block code. In our case, the encoder works *off-line* and thus can be implemented in software, whereas the decoder works *on-line* and thus must to be implemented in hardware.

Typically, block codes consist of codewords of fixed length. Systematic block codes are codes whose codewords have two parts: a fixed length information portion $Y \in \mathbb{F}_q^k$ and a fixed length redundant portion S. Our case is different; since Y represents a basic block its length depends on the number of instructions within a basic block and is not fixed. It is possible to work around this problem by adding jumps and NOPs to the program, but this in turn adds latency and increases the tamper-resistant memory size. In order to minimize the cost, the tag function f must be able to handle (in run-time) a q-ary vector Y of arbitrary length k,

$$k \le k_{max} = \lceil \frac{8 \cdot N}{r} \rceil,$$

without knowing its length beforehand.

C. Security metric

Recall that Y is stored in a regular memory. Thus, the attacker can alter it at will and even *change its length*. By contrast, the signature is not observable to the attacker and hence cannot be altered. Formally, denote by $\hat{Y} = (\hat{y}_{\hat{k}}, ...\hat{y}_1)$ the (possibly erroneous) sequence as read by the control flow checker. Note that the length of this sequence is $\hat{k}$, $1 \le \hat{k} \le k_{max}$. $\hat{k}$ may be *smaller, equal to, or greater* than k. Consequently, the checker "sees" the tuple $(\hat{Y}, X, f(X,Y))$ and has to decide whether this tuple is a codeword. Specifically, it computes $f(X,\hat{Y})$ and raises a flag if the computed value differs from the one stored in the tamper-resistant memory; that is, if $f(X,\hat{Y}) \neq f(X,Y)$. Therefore, we assess the effectiveness of this type of CFC as the probability Q that a precise attack will pass unnoticed. That is,

Definition 1 (Security metric.)**.** *Let X be a uniformly distributed random vector over a subset $\mathcal{X} \subseteq \mathbb{F}_q^t$, then the error masking probability is*

$$\bar{Q} = \max_{Y,\hat{Y}} Prob\left(f(X,Y) = f(X,\hat{Y}) \mid Y, \hat{Y}\right).$$

The function $f(X,Y)$ must be a nonlinear function in X and Y. Otherwise, if f can be written as $f(X,Y) = f_1(Y) + f_2(X)$, an attacker who knows Y and can choose which bits to flip in order to replace it by $\hat{Y}$ will choose a $\hat{Y}$ for which $f_1(Y) = f_1(\hat{Y})$; such an attack will never be detected. The following example clarifies this statement:

Example 2. *Let $q = 2$ and let Y be the basic block to be protected and denote by Y_0 the vector Y padded with $k_{max} - k$ zeros, $Y_0 = (\mathbf{0}_{k_{max}-k}, Y)$.*

Let $\mathcal{C}$ be a linear code of length $k_{max} + t + r_b$ bits and dimension $k_b = k_{max} + t$ defined by a (systematic) generator matrix

$$G = \left(\begin{array}{cc|c} I_{k_{max}\times k_{max}} & \mathbf{0}_{k_{max}\times t} & A_{k_{max}\times r_b} \\ \mathbf{0}_{t\times k_{max}} & I_{t\times t} & B_{t\times r_b} \end{array}\right).$$

A codeword in $\mathcal{C}$ is a triplet $(Y_0, X, f(X,Y_0)) = (Y_0, X)G$. The signature associated with Y is then $S = (X, f(X,Y))$ where the tag $f(X,Y)$ is $(Y_0, X)\cdot(A,B)^T$.

It is reasonable to assume that the tag length is smaller than the maximal basic block length; i.e., $k_{max} > r_b \ge rank(A)$. Therefore, for every Y there exists at least one vector $\hat{Y}$ for which $Y_0 A = \hat{Y}_0 A$. ($\hat{Y}$ and Y may be of different lengths). Since for every X, we have $\hat{S} = (X, \hat{Y}_0 A + XB) = S$ an attacker can replace Y by this $\hat{Y}$ without being detected. □

Numerical examples of attacks that can never be detected by linear codes are presented in the Appendix (Examples 12,13).

IV. Construction

A. Formal description of the tag function

We use a tag function f based on weakened version of the Karpovsky-Wang Algebraic Manipulation Detection (AMD) codes [15]. The computational complexity of f is smaller than Karpovsky-Wang's code since it makes use of the fact that the signature cannot be tampered with. This fact enable us to construct *a variable length code* whose checker can work in parallel to the execution of the basic block; the computed signature is ready before the last instruction of the current basic block leaves the pipeline. This makes this CFC an add-on module because it does not change the throughput or latency of the system. It also enables a simple and smooth transition between basic blocks.

One of the AMD codes presented in [15] relies on the Generalized Reed-Muller (GRM) codes. A GRM code is a *non-systematic fixed-length* code; it is defined by three parameters, r, t and b [20]:

- r - defines the size of the finite field ($q = 2^r$)
- t - is the number of random q-ary symbols
- b - is the order of the code, $b \le t(q-1)$

Table I: $\lambda(3,b)$ versus b

b	1	2	3	4	5	6	7	8
$\lambda(3,b)$	4	10	20	35	56	84	120	165

b is the smallest integer for which the coding scheme can protect a sequence of maximal length k_{max}; i.e., b is the smallest integer for which $k_{max} \leq \lambda(t,b)-1$ where

$$\lambda(t,b) = \sum_{j=0}^{t}(-1)^j \binom{t}{j}\binom{t+b-jq}{b-jq}.$$

In a GRM code, an information word Y of length k_{max} determines the coefficients of a polynomial of order $\leq b$. The corresponding GRM codeword is then a q-ary vector of length q^t whose symbols are the values that this polynomial takes.

Our code is built on the GRM code in the sense that we use a specific subset of the GRM codewords, and define $f(X,Y)$ as the value of the X'th symbol in the GRM codeword associated with the information word $Y_0 = (\mathbf{0}_{k_{max}-k}, Y)$. In what follows we define the mapping we use between Y and the polynomial. We start with several definitions.

Let $\mathbb{Z}_q$ be the set of integers $\{0,1,...,q-1\}$. Let $\mathbf{w} = (w_t, ...w_1) \in \mathbb{Z}_q^t$ be a q-ary vector of length t that represents the number $N(\mathbf{w}) = \sum_{j=1}^{t} w_j q^{j-1}$ in radix q. When it is clear from the context, we refer to a vector $\mathbf{w}$ by its value $N(\mathbf{w})$.

Define Ω_b to be an ordered set of size k_{max} of vectors whose sum is smaller or equal to b. That is,

$$\Omega_b = \{\mathbf{w}_i = (w_{i,t}, ...w_{i,1}) \in \mathbb{Z}_q^t \ : \ 0 < \sum_{j=1}^{t} w_{i,j} \leq b\}_{i=1}^{k_{max}}.$$

In addition, we require that the numbers associated with the vectors in Ω_b be the smallest numbers that have this property. In other words, $N(\mathbf{w}_1) = 1$, $N(\mathbf{w}_i) < N(\mathbf{w}_{i+1})$ and there is no other vector, say $\hat{\mathbf{w}} \notin \Omega_b$ such that $N(\mathbf{w}_i) < N(\hat{\mathbf{w}}) < N(\mathbf{w}_{i+1})$.

Example 3. *Let $t = 3, r = 8$. The values $\lambda(3,b)$ are given in Table I. For $b = 8$ the set Ω_b consists of 164 vectors,*

$$\left\{\begin{array}{cccccccc} - & (0,0,1) & (0,0,2) & (0,0,3) & \dots & (0,0,7) & (0,0,8) \\ (0,1,0) & (0,1,1) & (0,1,2) & (0,1,3) & \dots & (0,1,7) & - \\ (0,2,0) & (0,2,1) & (0,2,2) & (0,2,3) & \dots & - & - \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ (6,2,0) & - & - & - & \dots & - & - \\ (7,0,0) & (7,0,1) & - & - & \dots & - & - \\ (7,1,0) & - & - & - & \dots & - & - \\ (8,0,0) & - & - & - & \dots & - & - \end{array}\right\}$$

For example, $\mathbf{w}_{56} = (1,1,3)$ and $N(\mathbf{w}_{56}) = 1\cdot(2^8)^2 + 1\cdot(2^8)^1 + 3\cdot(2^8)^0 = 65795$. □

Construction 1 (The tag function f)**.** *Let Y be a q-ary vector of length $k \leq k_{max}$. Denote by $X^{\mathbf{w}}$ the product term*

$$X^{\mathbf{w}} = x_t^{w_t} \cdots x_2^{w_2} \cdot x_1^{w_1},$$

where computations are performed over $\mathbb{F}_q$. A polynomial based non-linear signature of Y is a binary vector of size $(t+1)\cdot r$ bits isomorphic to the vector $S = (X, f(X,Y)) \in \mathbb{F}_q^{t+1}$ where

$$f(X,Y) = \sum_{i=1}^{k} y_i X^{\mathbf{w}_i}, \tag{1}$$

and $\mathbf{w}_i \in \Omega_b$

Note that $f(X,Y) = f(X,Y_0)$ is the X'th symbol in a GRM codeword

$$c = (f(0,Y_0), f(1,Y_0), \dots, f(q^t-1, Y_0))$$

associated with the information symbol $Y_0 = (\mathbf{0}_{k_{max}-k}, Y)$.

Example 4. *Let the maximal length of a sequence be $N = 161$ bytes. Assume we want to design a control flow checker whose signature is a binary vector of length $32 = (3+1)\cdot 8$; that is, $r = 8$ and $t = 3$. Then we have $k_{max} = \lceil \frac{8\cdot 161}{8} \rceil = 161$, and $b = 8$ is the smallest integer for which*

$$\binom{3+b}{b} - 1 = 164 \geq k_{max}.$$

A signature is a binary vector of length 32 of the form $S = (X = (x_1,x_2,x_3), f(X,Y))$ where x_1, x_2, x_3 and f are 8 bit vectors that represent elements from the finite field $\mathbb{F}_{2^8}$. The function f for $k = 3$ is

$$\begin{aligned} f(X,Y) &= y_1 X^{(0,0,1)} + y_1 X^{(0,0,2)} + y_1 X^{(0,0,3)} \\ &= y_1 x_1 + y_1 x_1^2 + y_1 x_1^3, \end{aligned}$$

whereas for $k = k_{max}$ it is

$$\begin{aligned} f(X,Y) &= y_1 X^{(0,0,1)} + y_2 X^{(0,0,2)} + ...y_8 X^{(0,0,8)} + \\ &\quad y_9 X^{(0,1,0)} + y_{10} X^{(0,1,1)} + ...y_{16} X^{(0,1,7)} + \\ &\quad \vdots \\ &\quad y_{159} X^{(6,0,1)} + y_{160} X^{(6,1,0)} + y_{161} X^{(7,0,0)} \\ &= y_1 x_1 + y_2 x_1^2 + ...y_8 x_1^8 + \\ &\quad y_9 x_2 + y_{10} x_2 x_1 + ...y_{16} x_2 x_1^7 + \\ &\quad \vdots \\ &\quad y_{159} x_3^6 x_1 + y_{160} x_3^6 x_2 + y_{161} x_3^7. \end{aligned}$$

□

B. The effectiveness of the CFC

It is important to note that the triplet $(Y, X, f(X,Y))$ is a codeword in an error detecting code; it has no error correction capabilities. Thus, the decoder has no error-recovery mechanism. This serves to avoid scenarios in which an attacker can manipulate the system and make the decoder conceal the attack by "correcting" an erroneous basic block into a legal but different basic block. Example 13 in the Appendix shows how simple it is to manipulate a system when the error correction mechanism is activated.

The following theorem provides an *upper bound* on the probability that the CFC will not detect a tampered-with basic block. The bound applies to all basic blocks regardless of their length and content and therefore obviates the need for experiments/simulations.

Theorem 1. *Let X be a random vector that is uniformly distributed over $\mathcal{X} \subseteq \mathbb{F}_q^t$. The probability that a GRM based signature will not detect a tampered sequence is*

$$\bar{Q} \leq \frac{bq^{t-1}}{|\mathcal{X}|}.$$

Proof. Let Y be a q-ary vector of length k that represents the correct sequence, and denote by $\hat{Y}$ the q-ary vector of length $\hat{k}$ that represents the tampered sequence. The two sequences may be of different lengths, i.e., $k \neq \hat{k}$. Notice that the expansion of Y and $\hat{Y}$ into q-ary vectors of length k_{max} does not change the signature since $f(X,Y) = f(X,(\mathbf{0}_{k_{max}-k}, Y))$ and $f(X,\hat{Y}) = f(X,(\mathbf{0}_{k_{max}-\hat{k}}, \hat{Y}))$. Thus, without loss of generality, we assume that both vectors are of size k_{max}. This enables us to represent $\hat{Y}$ as

$$\hat{Y} = Y + E$$

where $Y, \hat{Y}$ and E are vectors in $\mathbb{F}_q^{k_{max}}$. E is defined as the difference between the two sequences and hence can be treated as an additive error vector. The error masking probability is then

$$\bar{Q} = \max_{E \in \mathbb{F}_q^{k_{max}} \setminus \{\mathbf{0}\}} Q(E).$$

The error E is detected if the computed signature of $\hat{Y}$ differs from the signature of Y. In other words, the attack is undetected if $f(Y,X) = f(Y+E,X)$. Define,

$$g_E(X) = f(Y,X) - f(Y+E,X) = \\ = \sum_{i=1}^{k} y_i X^{\mathbf{w}_i} - \sum_{i=1}^{k} (y_i + e_i) X^{\mathbf{w}_i} = \sum_{i=1}^{k} e_i X^{\mathbf{w}_i}. \quad (2)$$

Then, a nonzero E is undetected if X is a root of the polynomial g_E. This polynomial is associated with a q-ary codeword c of length q^t in the generalized Reed-Muller (GRM) code. That is,

$$c = (g_E(0), g_E(1), ..., g_E(q^t - 1)) \neq \mathbf{0}_{q^t}.$$

Since the GRM is a linear code of minimum distance $d = (q-b)q^{t-1}$, every nonzero codeword c has a minimal weight d. That is, g_E has at most $q^t - d$ roots. Hence, for a uniformly chosen non-zero vector $X \in \mathcal{X}$, the probability that tampering will go undetected is

$$Q(E) \leq \frac{q^t - d}{|\mathcal{X}|} = \frac{bq^{t-1}}{|\mathcal{X}|}.$$

□

Example 5. *Consider the code in Example 4. The code has $t = 3, b = 8$ and $r = 8$, hence for $|\mathcal{X}| = (2^r)^t = 2^{24}$, the probability that an attack will be masked is approximately 2^{-5}.*

Note that another way to construct a control flow checker for $N = 161$ bytes is by taking a larger r; i.e., $r = 16, t = 1$ and $b = \lceil N/2 \rceil = 81$. In this case, the signature is a binary vector of length $(1+1)\cdot 16$ bits, and f is a polynomial of a single variable, $f(X,Y) = y_1 x + y_2 x^2 + \cdots + y_{81} x^{81}$. Here, the computation is performed in the (larger) field $\mathbb{F}_{2^{16}}$; hence, the implementation cost is larger, but the probability that an attack will be masked becomes significantly smaller ($\approx 2^{-9}$). □

Table II shows several constructions for different block and signature sizes. The first column lists the length of the maximal sequence (in bytes), the probability Q that an attack will be masked with

$$\mathcal{X}_1 = \{X = (x_t, \ldots, x_1) \ : \ x_i \neq 0 \ \forall i\} \subset \mathbb{F}_q^t$$

and with $\mathcal{X}_2 = \mathbb{F}_q^t$, in the second and third columns, respectively. The signature size (in bits) is appears in the fourth column, and the GRM parameters, r, t and b are given in columns 5-7.

The analysis of the error masking probability $\bar{Q}$ is a worst case analysis. The error masking probability can be smaller when the g_E is of low degree. The following example illustrates this statement:

Example 6. *Assume $X = (x_3, x_2, x_1) \in \mathcal{X}$, and let*

$$g_E = e_1 x_1 + e_2 x_1^2 + e_3 x_1^3 x_2.$$

For a given $(x_3, x_2 \neq 0)$ pair, g_E is a polynomial of degree 3, and for pairs $(x_3, x_2 = 0)$ it is of degree 2. Hence, g_E has at most $(3q(q-1) + 2q)$ roots. The probability that this error is masked is:

$$Q(E) \leq \frac{q(3q-1)}{|\mathcal{X}|} < \frac{8q^2}{|\mathcal{X}|}.$$

□

V. CFC DESIGN FOR A 32-BIT SINGLE-PIPELINE PROCESSOR WITH A 32-BIT SIGNATURE

In this section we detail the design of a CFC protecting a single-pipeline processor with a 32 bit ISA from malicious attacks. We assume that the pre-computed signatures are stored in a tamper-resistant memory. We start by describing the considerations that underlie the choice of design parameters, then describe the architecture of the computational module and elaborate on the structure of each of its blocks, with a focus on the correctness of the implementation rather than its efficiency. Finally, we describe the architectural changes needed to make it an effective real-time CFC.

Table II: Code parameters for signature size ≤ 32 bits

max N (bytes)	Q with $\mathcal{X}_1$	Q with $\mathcal{X}_2$	Signature size (bits)	r	t	b
156	0.0998	0.0938	30	6	4	6
143	0.0640	0.0625	28	7	3	8
164	0.0316	0.0313	32	8	3	8
160	0.0127	0.0127	30	10	2	13
370	0.1331	0.1250	30	6	4	8
315	0.0880	0.0859	28	7	3	11
261	0.0186	0.0186	30	10	2	19
220	0.0352	0.0351	32	8	3	9
535	0.1498	0.1406	30	6	4	9
594	0.1120	0.1094	28	7	3	14
527	0.1220	0.1211	24	8	2	31
559	0.0514	0.0508	32	8	3	13
522	0.0569	0.0566	27	9	2	29
542	0.0274	0.0273	30	10	2	28
1364	0.1997	0.1875	30	6	4	12
1139	0.0672	0.0664	32	8	3	17
1075	0.0391	0.0391	30	10	2	40

Table III: Code parameters for $r = 8$

max N (bytes)	Q with $\mathcal{X}_1$	signature (bits)	t	b
62	0.0394	24	2	10
69	0.0159	40	4	4
83	0.0237	32	3	6
135	0.0591	24	2	15
220	**0.0352**	**32**	**3**	**9**
164	**0.0316**	**32**	**3**	**8**
209	0.0238	40	4	6
275	0.0866	24	2	22
285	0.0395	32	3	10
329	0.0278	40	4	7

A. Design parameters

All the code parameters are linked: as we saw in the previous section, the triplet t, b and r affects N, the code's error masking probability $\bar{Q}$, and the signature size. In what follows, we describe the design considerations that led to the choice of N and r, which in turn determine parameters b and t for the selected processor and the 32-bit signature.

1) The maximal basic block length N : The parameter N represents the maximal basic block length (in bytes) that can be protected by the code. Since any basic block can be split into several basic blocks of smaller length, it is assumed that the encoder and the checker "see" basic blocks of length smaller or equal to N.

Note that splitting a basic block larger than N-bytes into smaller basic blocks requires additional rows in the signature memory; hence, the maximal basic block size cannot be too small. On the other hand, a large N may increase the time between the execution of a tampered instruction and its detection. Therefore, N cannot be too large either.

The basic block size, along with the execution of a jump instruction, indicates the end of a basic block. To cope with a case where a basic block ends with a label (see Example 1), we can keep the size of the basic block in the tamper-resistant memory, as part of the signature, or insert an unconditional jump to the next (labeled) instruction at the end of this basic block. Figure 3, taken from [21], shows the distribution of BB sizes for some real Linux-based applications (*ghostscript, head, hexdump, sort, tail*) running on a x86 architecture. As can be seen, the vast majority of BBs have a number of bytes that is smaller than

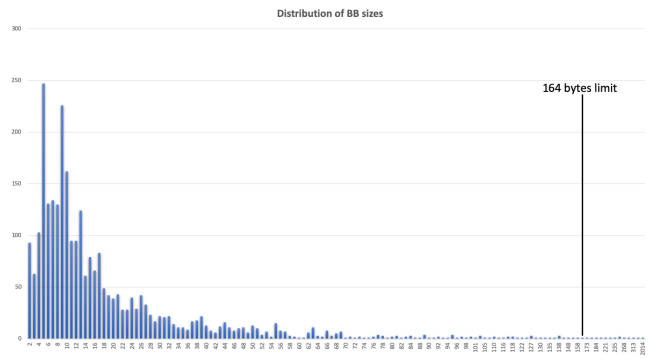

Figure 3: Sizes of Basic Blocks taken from some real Linux-based applications [21].

Table IV: Code parameters for $r \geq 8$

r	max N (bytes)	Q with $\mathcal{X}_1$	signature (bits)	t	b
8	**220**	**0.0352**	**32**	**3**	**9**
8	**164**	**0.0316**	**32**	**3**	**8**
8	209	0.0238	40	4	6
9	133	0.0275	27	2	14
9	133	0.0138	36	3	7
10	130	0.0127	30	2	13
10	148	0.0069	40	3	7
11	143	0.0064	33	2	13
12	135	0.0029	36	2	12

$N = 164$, thus confirming that the choice of these parameters is reasonable.

2) The tag size r: For simplicity, we work with bytes ($r = 8$). That is, both the y's and the x's are elements of $\mathbb{F}_{2^r}$ (that is, isomorphic to $\mathbb{F}_2^r$). Table III shows several constructions for $r = 8$ ($q = 2^8$). The columns from left to right are the maximal basic block length (N), the probability Q that an attack will be masked, the signature size in bits, and the GRM parameters, t and b.

In fact, r can take larger values (i.e., $r > 8$). In this case the x's should take a nonzero value from $\mathbb{F}_{2^r}$ and the eight bits of the y's should be padded with $r - 8$ zeros. This may lead to constructions with a smaller error masking probability with a similar implementation cost. Table IV shows the error masking probability for several r values and for different (maximal) lengths of basic blocks. Note that the size of the signature, which is a function of r and t, takes values from 27 to 40. It is clear from the table that by adding a *single bit* to the signature, one can implement a checker with $r = 11$ and obtain a smaller error masking probability of 0.0064 instead of 0.0316.

Guided by these design considerations, we implemented a CFC with the following parameters (marked in bold in Table IV):

- **Serial implementation (Section V-B)**

$$r = 8,\ t = 3,\ b = 8,\ N = 164, \text{bytes}\ q = 256,$$

and error masking probability $\bar{Q} = 8 \cdot q^2/(q^3 - 1) = 3.13\%$.

- **Parallel implementation (Section V-F)**

$$r = 8,\ t = 3,\ b = 9,\ N = 144, \text{bytes}\ q = 256,$$

and error masking probability $\bar{Q} = 9 \cdot q^2/(q^3 - 1) = 3.5\%$.

B. Generic architecture of the Computation Module (CM)

A simplified block diagram of the Computation Module (CM) is shown in Fig. 4. It receives the random portion X of the current basic blockfrom the tamper-resistant memory and the content of the basic block as fetched from the instruction cache. The CM has its own clock, whose frequency depends on the number of y symbols processed each clock period. For simplicity, we first describe the operation of the CM when it receives one byte per clock and in Section V-F we show how to use this simplified design to process four bytes per clock. In the latter case, the system's clock is used as the CM's clock.

The CM consists of three modules:

- **A (t, b) counter.** These counters are used for generating the t-digit vectors in Ω_b. The inputs to this block are the CM clock and the global reset signals from the control unit. The current state of the counters $\mathbf{w} \in \Omega_b$ is used to compute internal control signals (inc_j and $reset_j$, $j = 1, \ldots t$) that determine its next state. The t-bit control signal $inc = (inc_t, ..., inc_1)$ is also used as input to the product term computation block (described next).
- **A product term generator.** This block computes $X^{\mathbf{w}}$. The inputs to this block are the CM clock, the global reset signals from the control unit, the secret key - X, and the control signals from the counter block. The block consists of t r-bit registers dubbed $R_1, \ldots R_t$ and a single finite field multiplier. Given these registers and the fact that the Ω_b is an ordered set, $X^{\mathbf{w}}$ can be computed without using $\mathbf{w}$. That is, $\mathbf{w}$ is an internal variable of the counter/s. Thus, only t wires connect the counter and this block (instead of $t \cdot log_2(b)$).
- **A polynomial evaluator.** This block computes $f(Y, X)$.

In what follows we elaborate on each module.

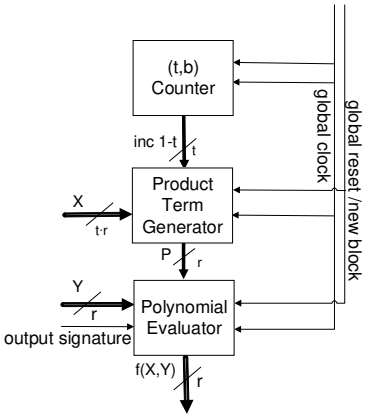

Figure 4: Generic architecture of the computation module

C. The (t, b) Counter module

The (t, b) counter is a state-machine; it produces the vectors Ω_b in an ordered manner. At time slot i the counter generates the vector

$$\mathbf{w}_i = (w_{i,t}, \ldots, w_{i,2}, w_{i,1}) \in \Omega_b.$$

A schematic block diagram of the (t, b) counters is depicted in Fig. 5 and its formal description is given in Alg. 1.

The counter can be viewed as a radix $q = 2^r$ ripple counter. For example, for $t = 3, r = 8, b = 8$, if the current vector $\mathbf{w}$ is $(w_3 = 5, w_2 = 0, w_1 = 1)$, the next vector will be $(5, 0, 2)$. However, it differs from a conventional t-digit counter in that a conventional counter has a global reset/preset signal that initializes (simultaneously) all the digits to a predefined value, whereas in our case the counter has t local reset and increment signals, $reset_j$ and inc_j (Alg.1, Line 11). Thus it can increment its upper part, e.g., $(w_t, \ldots, w_{j+1})$, and reset its lower part $(w_j, \ldots, w_1)$, see Alg.1, Lines 7-10. For example, $(4, 2, 2)$ will be followed by

$$\mathbf{w}_{i+1} = (4, 3, 0),\ \mathbf{w}_{i+2} = (4, 3, 1)\ \mathbf{w}_{i+3} = (4, 4, 0).$$

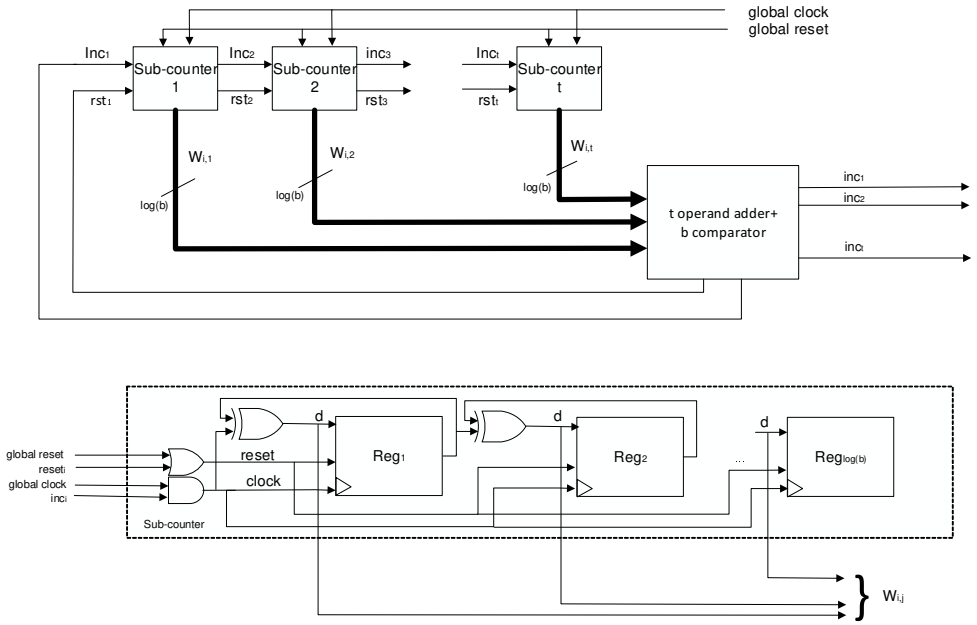

Figure 5: The (t, b) counter module: The top figure shows how the t sub-counters are connected to each other and the bottom figure shows a schematic design of a sub-counter.

Algorithm 1 (t, b) Counter for $b < q$

1: (Initialization step) Set $i = 1$.
2: (Initialization step) Set $\mathbf{w}_i = (0, 0, \ldots, 0, 1)$.
3: **while** the basic block has not ended, **do**
4: $inc_1 := (\sum_{j=1}^{t} w_{i,j} < b)$
5: $reset_1 := \neg inc_1$
6: **for** $j = 1 : t$ **do**
7: Update the value of the j'th digit $w_{i+1,j}$:
8: if $(reset_j == 1)$ then $w_{i+1,j} := 0$,
9: if $(reset_j == 0) \wedge (inc_j == 0)$ then $w_{i+1,j} := w_{i,j}$,
10: if $(reset_j == 0) \wedge (inc_j == 1)$ then $w_{i+1,j} := w_{i,j} + 1$,
11: Generate the reset and increment signals for the $(j + 1)$'th digit:
12: $reset_{j+1} := (reset_j == 1) \wedge (w_{i,j} == 0)$,
13: $inc_{j+1} := (reset_j == 1) \wedge (w_{i,j} > 0)$.
14: **end for**
15: $i = i + 1$ // Virtual counter to simplify the analysis
16: **end while**

Note that $N(4, 2, 2) < N(4, 3, 0) < N(4, 3, 1)$.

Fig. 5-top shows a (t, b) composed of t sub-counters that work in parallel. Each sub-counter produces r bit vectors, dubbed *'digit'*. Each sub-counter has its own reset and increment signals that are generated by the preceding sub-counter (Alg.1, Lines 11-13). The first sub-counter is controlled by the b comparator (Alg.1, Lines 4-5). The bottom figure shows a detailed scheme of a sub-counter. At each cycle, if the increment signal is raised, the sub-counter increments its value by 1 (Alg.1, Line 10). The first sub-counter, cnt_1, is set to 1 on a global reset (Alg.1, Line 2). All the other sub-counters can be set to zero by either external reset (Line 2) or by a reset from the preceding sub-counter (Line 12).

Theorem 2. *For $b < q$, Alg. 1 generates all the vectors in Ω_b in increasing order.*

The proof of this theorem is given in the Appendix.

D. Product term computation module

This block computes the product term $P_i = X^{\mathbf{w}_i}$. The schematics of this module are shown in Fig 6. Alg. 2 explains how it operates and the associated theorem proves its correctness.

The block consists of one finite field multiplier and t "shadow" registers, where each register R_j holds a different product. Recall that the product term computation module does not "see" the value of $\mathbf{w}_i$, and thus has to figure out how to compute the current product P_i by analyzing the t bits of the inc vector. Denote by j^+ the largest index for which $inc_j = 1$ (Alg.2, Line 3); the i'th product term is computed by multiplying the value stored in the (j^+)'th shadow register by x_{j^+} (Alg.2, Line 4). Then, the value of the first shadow registers, $R_1, \ldots R_{j^+}$ is updated with the new product P_i (Line 5).

Theorem 3. *Algorithm 2 correctly computes $P_i = X^{\mathbf{w}_i}$ from the signals $inc_1, \ldots inc_t$.*

The proof of this theorem is given in the Appendix.

Algorithm 2 Compute Product Term

1: (Initialization step) Set $R_j = 1$ for all $1 \leq j \leq t$.
2: **while** the basic block has not ended, **do**
3: Find the largest index j^+ for which $inc_{j^+} == 1$.
4: Compute $P_i := R_{j^+} \cdot x_{j^+}$.
5: **for** $j = 1 : j^+$ **do**
6: Update the register with the current product term: $R_j := P_i$, $load_j := 1$.
7: **end for**
8: $i = i + 1$ // Virtual counter to simplify the analysis
9: **end while**

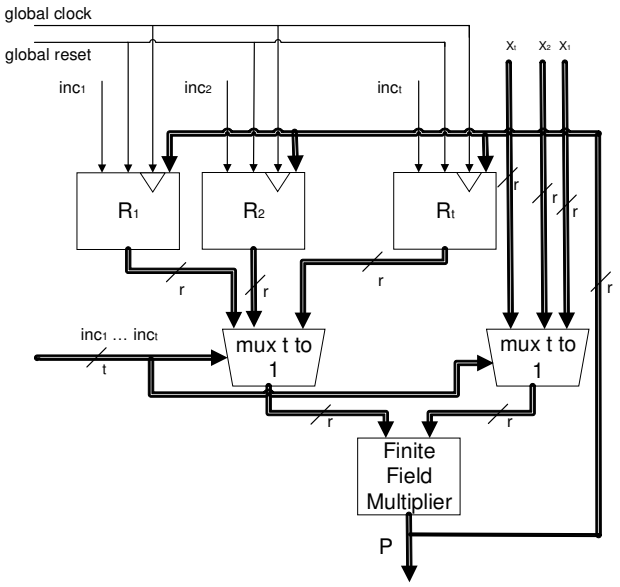

Figure 6: Product term computation module. As described in Alg. 2, at each cycle the value of R_{j^+} is multiplied by x_{j^+}. The value of each register $R_1, ..., R_{j^+}$ is updated with the value of the product P.

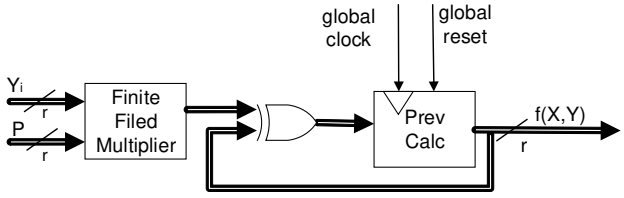

Figure 7: Polynomial Evaluation module.

Example 7. *Consider the CFC in Example 4. Table V shows the content of the shadow registers over time.* □

E. Polynomial Evaluation module

The polynomial $f(Y, X)$ can evaluated at the same time as the generation of the vectors in Ω_b. As described at Fig. 7, it consists of a multiplier and an adder. The computed P_i is multiplied by the corresponding y_i and the result is added to the content of the register that holds the sum of the products computed so far.

F. Concurrent CFC implementation

A schematic block diagram of the checker is depicted in Fig 8. Every clock cycle, 32 bits are read from the memory; therefore, at every clock cycle, $4 = 32/r$ q-ary symbols, $y_i, y_{i+1}, \ldots y_{i+3}$, enter the checker. As shown in the figure, there are four different (t, b) counters: the j'th counter has t_j digits with design parameter b_j. Each counter can have its own initialization value (or values).

The output of the j'th counter at time i, $\mathbf{w}_{i,j}$, enters a product term computation module that generates the product $P_{i,j}$ where $P_{i,j} = X_j^{\mathbf{w}_{i,j}}(X \setminus X_j)^{\mathbf{v}_j}$. Here, $X_j \subseteq X$ is the predefined t_j q-ary random variables allocated to the j'th counter, $(X \setminus X_j)$ are the remaining variables, and $\mathbf{v}_j \in \mathbb{Z}^{t-t_j}$ is a predefined integer vector whose L^1 norm is equal to or smaller than $b - b_j$.

The polynomial evaluation module in Fig. 8 computes the sum $\sum_{j=1}^{4} y_{i+j-1} P_{i,j}$ and adds it to the sum accumulated so far.

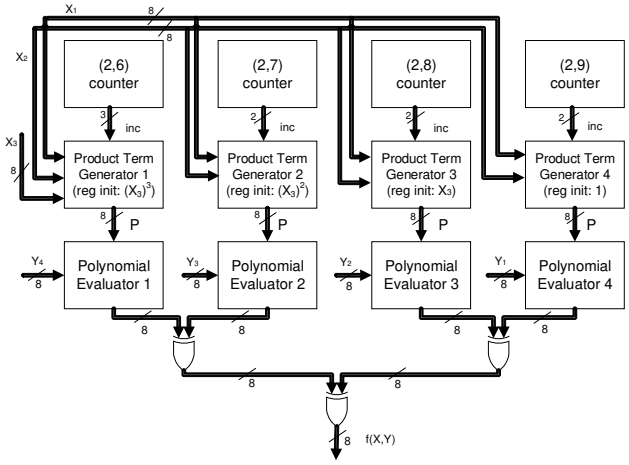

Figure 8: Control flow checker architecture for a 32-bit processor and $r = 8$.

Four *disjoint sets* of $\mathbf{w}$'s are generated by the counters. Denote the sets as $\Omega_9^{(0)}, \Omega_9^{(1)}, \Omega_9^{(2)}$ and $\Omega_9^{(3)}$. These sets satisfy $\Omega_9^{(j)} \cap \Omega_9^{(l)} = \Phi$ for $j \neq l$, and $|\cup_j \Omega_9^{(j)}| \geq N$. The signature value is then

$$f(X, Y) = \sum_{i=1}^{k/4} \sum_{j=0}^{3} y_{4i+j,j} X^{\mathbf{w}_{i,j}}, \tag{3}$$

where $y_{4i+j,j}$ is the j'th byte of the i'th instruction in a basic block that has k bytes, and $\mathbf{w}_{i,j}$ is the i'th vector in the ordered set $\Omega_9^{(j)}$.

There are several ways to split Ω_9 into four disjoint sets. One example is the following:

$$\begin{aligned} \Omega_9^{(0)} &= \{\mathbf{w} \ : \ \mathbf{w} \in \Omega_9 \text{ and } w_3 = 0\} \\ \Omega_9^{(1)} &= \{\mathbf{w} \ : \ \mathbf{w} \in \Omega_9 \text{ and } w_3 = 1\} \\ \Omega_9^{(2)} &= \{\mathbf{w} \ : \ \mathbf{w} \in \Omega_9 \text{ and } w_3 = 2\} \\ \Omega_9^{(3)} &= \{\mathbf{w} \ : \ \mathbf{w} \in \Omega_9 \text{ and } w_3 \geq 3\}. \end{aligned} \tag{4}$$

Note that the sets are of different sizes $|\Omega_9^{(0)}| = 54$, $|\Omega_9^{(1)}| = 45$, $|\Omega_9^{(2)}| = 36$, $|\Omega_9^{(3)}| > 40$. This implies that the maximal Basic Block must be smaller or equal to $4 \cdot 36 = 144$ bytes. This number is smaller than 164 which is the maximal basic block size that can be supported by an 8-bit architecture.

Recall that the four sets are generated by four counters that work in parallel. Table VI presents the parameters of these counters. For each counter, the table specifies the parameter vector (t_i, b_i), the maximal number of $\mathbf{w}$'s that it can generate and the form of the corresponding product. Table VII shows the operation of the four counters and the control signal they generate at different time slots. Note that the first counter starts from 1 whereas the other three start from zero.

Since N is determined by the size of the smallest counter (in our case $\Omega_9^{(2)}$), we can use its saturation signal; i.e., its inc_2 signal, to notify the control unit that the current basic block has ended.

It is important to note that the product terms that correspond to elements in $\Omega_9^{(0)}$ are of the form $x_3^0 \hat{X}^{\hat{\mathbf{w}}}$ where $\hat{X} = x_2 x_1$ and $\hat{\mathbf{w}}$ are the vectors associated with a smaller code with parameters $r = 8, t = 2$ and $b = 9$. Similarly, the product terms corresponding to elements in $\Omega_9^{(2)}$ are of the form $x_3^2 \hat{X}^{\hat{\mathbf{w}}}$ where the $\hat{\mathbf{w}}$'s are associated with code with parameters $r = 8, t = 2, b = 7$. Overall, a checker for this code will consist of four different encoders of smaller codes and will have to multiply each partial product $\hat{X}^{\hat{\mathbf{w}}}$ by a different power of x_3 (see Fig. 8). Specifically, additional finite field multipliers are required to compute, for example, x_3^3.

Table V: The content of the shadow registers in Example 4

i	1	2	3	...	9	10	11	...	44	45	46	47	...	54	55	...
w_i	(0,0,1)	(0,0,2)	(0,0,3)	...	(0,1,0)	(0,1,1)	(0,1,2)	...	(0,8,0)	(1,0,0)	(1,0,1)	(1,0,2)	...	(1,1,1)	(1,1,2)	...
inc	(0,0,1)	(0,0,1)	(0,0,1)	...	(0,1,0)	(0,0,1)	(0,0,1)	...	(0,1,0)	(1,0,0)	(0,0,1)	(0,0,1)	...	(0,0,1)	(0,0,1)	...
R_1	1	x_1	x_1^2	...	x_1^8	x_2	x_2x_1	...	$x_2^7x_1$	x_2^8	x_3	x_3x_1	...	x_3x_2	$x_3x_2x_1$	...
R_2	1	1	1	...	1	x_2	x_2	...	x_2^7	x_2^8	x_3	x_3	...	x_3x_2	x_3x_2	...
R_3	1	1	1	...	1	1	1	...	1	1	x_3	x_3	...	x_3	x_3	...

Table VI: Counter parameters and initialization for $r = 8, t = 3$ and $b = 9$, $Q = 0.035$

cnt #1	cnt #2	cnt #3	cnt #4
(2,9)	(2,8)	(2,7)	(2,6)
k=54	k=45	k=36	k=28
(0**)	(1**)	(2**)	(3**)
			(2,5)
			k=21
			(4**)

G. Operation

Consistent with Theorem 1, every error will be detected with a probability of at least $(1 - \bar{Q}) = 96.87\%$. In fact, for a given pair consisting of a basic block and a tampered-with block the exact probability that the CFC will not detect the error can be calculated, following the proof of the theorem.

In the Appendix we provide four examples that illustrating how the CFC works and how it detects an error:

- Example 8 shows how the CFC works when the tag is calculated in an error-free scenario.
- Example 9 shows a case where two instructions are erroneous.
- Examples 10 and 11 show the tag calculation when the attacker changes the size of a basic block.

H. Implementation cost

The coding scheme presented in this section was implemented for the chosen parameters. We synthesized the circuit by using a 28nm CMOS technology. The results of the synthesis led to an area occupancy of about 1700 Gate Equivalents (GEs). In order to compare our solution to existing ones, we calculated (when possible) the area of the other solutions in GEs. The values we obtained are sensitive to errors since not all the technological details are provided (nor units in some cases). For instance, in [6], the sizes are not explicitly calculated. However, they declare a 30% area overhead w.r.t. Leon3. Leon3 implementations range from 300K to 450K GE, thus leading to a rough approximation of at least 90K GEs for their implementation. Table VIII presents the comparison to the other works discussed in this paper. As can been seen, our solution has the smallest overhead while guaranteeing a high level of security.

Concerning the impact in terms of speed, the circuit can work at more than 1 GHz when working at 1.3V, thus confirming the possibility to run in parallel with modern processors without incurring in additional delay. It is important to note that we did not considered the impact of the tamper-resistant memory in this analysis. We have assumed that such a memory is available and can be easily integrated into the system. Example of existing tamper-resistant non-volatile memories are proposed in [22] and [23].

VI. Conclusion

This paper presents a non-linear encoder and checker that can be used in every Control Flow Checking mechanism. It suggests a way to design an add-on, low overhead, fine-grained checker with no need for architectural changes. Similar to other code-based solutions, it has the advantages of not introducing additional latency, has low overhead and is able to protect basic blocks of variable length. However, in contrast to existing code-based solutions, it employs non-linear codes. In this sense, it guarantees the high level of security of the MAC-based system, without introducing high area penalties. As in every MAC-based solution, the code integrates a secret part that must be stored (together with the pre-calculated signatures) in a tamper-resistant memory.

References

[1] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, (New York, NY, USA), pp. 13:1–13:6, ACM, 2014.

[2] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "A watchdog processor to detect data and control flow errors," in *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*, pp. 144–148, July 2003.

[3] A. Chaudhari, J. Park, and J. Abraham, "A framework for low overhead hardware based runtime control flow error detection and recovery," in *IEEE 31st VLSI Test Symposium (VTS), Berkeley, CA*, pp. 1–5, IEEE, 2013.

[4] J. Abraham and R. Vemu, "Control flow deviation detection for software security," Mar. 11 2010. WO Patent App. PCT/US2009/047,390.

[5] R. de Clercq and I. Verbauwhede, "A survey of hardware-based control flow integrity (CFI)," *CoRR*, vol. abs/1706.07257, 2017.

[6] R. d. 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 *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1172–1177, March 2016.

[7] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, pp. 1295–1308, Dec 2006.

[8] M. Werner, R. Schilling, T. Unterluggauer, and S. Mangard, "Protecting risc-v processors against physical attacks," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1136–1141, 2019.

[9] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey," *IEEE Transactions on Computers*, vol. 37, pp. 160–174, Feb 1988.

[10] K. Wilken and J. P. Shen, "Continuous signature monitoring: efficient concurrent-detection of processor control errors," in *International Test Conference New Frontiers in Testing*, pp. 914–925, Sep. 1988.

[11] M. Werner, E. Wenger, and S. Mangard, "Protecting the control flow of embedded processors against fault attacks," in *Smart Card Research and Advanced Applications* (N. Homma and M. Medwed, eds.), (Cham), pp. 161–176, Springer International Publishing, 2016.

[12] Y. Xie, X. Xue, J. Yang, Y. Lin, Q. Zou, R. Huang, and J. Wu, "A logic resistive memory chip for embedded key storage with physical security," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 63, pp. 336–340, April 2016.

[13] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 55–71, 2019.

[14] G. Di Natale, M. L. Flottes, S. Dupuis, and B. Rouzeyre, "Hacking the control flow error detection mechanism," in *IEEE*

Table VII: Counter values over time for concurrent CFC implementation
($b = 9$, $t = 3$, $r = 8$)

	i	1	2	3	…	6	7	8	9	10	…	27	28	29	…	34	35	36	…
cnt#1	$\omega_{i,1}$	1	2	3	…	6	7	8	9	0	…	0	1	2	…	0	1	2	…
(b=9)	$\omega_{i,2}$	0	0	0	…	0	0	0	0	1	…	3	3	3	…	4	4	4	…
	j^+	1	1	1	…	1	1	1	1	2	…	2	1	1	…	2	1	1	…
cnt#2	$\omega_{i,1}$	0	1	2	…	6	7	8	0	1	…	3	4	5	…	4	0	1	…
(b=8)	$\omega_{i,2}$	0	0	0	…	0	0	0	1	1	…	3	3	3	…	4	5	5	…
	j^+	-	1	1	…	1	1	1	2	1	…	1	1	1	…	1	2	1	…
cnt#3	$\omega_{i,1}$	0	1	2	…	6	7	0	1	2	…	1	2	3	…	1	0	1	…
(b=7)	$\omega_{i,2}$	0	0	0	…	0	0	1	1	1	…	4	4	4	…	6	7	0	…
	j^+	-	1	1	…	1	1	2	1	1	…	1	1	1	…	1	2	1	…
cnt#4	$\omega_{i,1}$	0	1	2	…	6	0	1	2	3	…	0	0	1	…	5	6	0	…
(b=6)	$\omega_{i,2}$	0	0	0	…	0	1	1	1	1	…	6	0	0	…	0	0	1	…
	j^+	-	1	1	…	1	2	1	1	1	…	2	**3**	1	…	1	1	2	…

Table VIII: Area Occupancy comparison

Solution	Area of the computation module [GEs]
Proposed	1700
[6]	90K
[15]	N/A (requires multipliers of 36, 60, 90 bits)
[16]	N/A (requires an AES cipher)
[18]	N/A (requires an RSA cipher)
[24]	3500

2nd International Verification and Security Workshop (IVSW), Thessaloniki, pp. 51–56, IEEE, 2017.

[15] Z. Wang and M. Karpovsky, "Algebraic manipulation detection codes and their applications for design of secure cryptographic devices," in *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International*, pp. 234–239, IEEE, 2011.

[16] A. M. Fiskiran and R. B. Lee, "Runtime execution monitoring (rem) to detect and prevent malicious code execution," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.*, pp. 452–457, Oct 2004.

[17] A. T. Abdalsatir and A. J. Abboud, "Integrity checking of several program codes," *Journal of Engineering and Applied Sciences*, 01 2019.

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

[19] C. Bresch, D. Hély, R. Lysecky, S. Chollet, and I. Parissis, "Trustflow-x: A practical framework for fine-grained control-flow integrity in critical systems," *ACM Trans. Embed. Comput. Syst.*, vol. 19, Sept. 2020.

[20] P. Delsarte, J. Goethals, and F. M. Williams, "On generalized reedmuller codes and their relatives," *Information and Control*, vol. 16, no. 5, pp. 403 – 442, 1970.

[21] G. D. Natale and O. Keren, "Nonlinear codes for control flow checking," in *2020 IEEE European Test Symposium (ETS)*, pp. 1–6, 2020.

[22] N. Rangarajan, S. Patnaik, J. Knechtel, O. Sinanoglu, and S. Rakheja, "Smart: A secure magnetoelectric antiferromagnet-based tamper-proof non-volatile memory," *IEEE Access*, vol. 8, p. 76130–76142, 2020.

[23] *DS3660 Datasheet*.

[24] K. Yumbul, S. S. Erdem, and E. Savas, "On protecting cryptographic applications against fault attacks using residue codes," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pp. 69–79, IEEE, 2011.

Gilad Dar received the B.S. degree in Computer Engineering from the Bar-Ilan University, Israel, in 2019. He is currently pursuing the master's degree with the Faculty of Engineering, Bar-Ilan University, Israel. His current research interests include coding theory, digital integrated circuit design, and side channel attacks and countermeasures.

Giorgio Di Natale received the PhD in Computer Engineering from the Politecnico di Torino (Italy) in 2003, and the HDR (Habilitation a Diriger les Recherches), in 2014 from the University of Montpellier II (France). He is Director of Research for the National Scientific Research Center of France at the TIMA laboratory, a joint research laboratory between the CNRS, the University of Grenoble-Alpes and Grenoble-INP. His research interests include hardware security and trust, secure circuits design and test, reliability evaluation and fault tolerance, software implemented hardware fault tolerance, and VLSI testing. He serves as chair of the TTTC of the IEEE Computer Society, he is Golden Core member of the Computer Society and Senior member of the IEEE.

Osnat Keren received the M.Sc. degree in electrical engineering from the Technion–Israeli Institute of Technology and the Ph.D. degree from Tel Aviv University, Israel. After working at High Tech for several years, in 2004, she took up a faculty position at the Faculty of Engineering, Bar-Ilan University, Israel.

VII. APPENDIX

A. Proof of Theorem 2

First we show that each generated vector is a member of Ω_b. The proof is by induction. Clearly, $\mathbf{w}_1 \in \Omega_b$. Assume that $\mathbf{w}_i \in \Omega_b$. If the sum of the digits of $\mathbf{w}_i$ is smaller than b (Alg. 1 line 4), the value of the least significant digit $w_{i+1,1}$ will be incremented by one (line 10). Consequently, the $reset_2$ and inc_2 signals of the second digit will be set to zero (line 12-13) and hence the second digit will keep its value (line 9). The same applies to all the other digits, and hence $w_{i+1,j} = w_{i,j}$ for all $j > 1$, and therefore, $\sum_{j=1}^{t} w_{i+1,j} = \sum_{j=1}^{t} w_{i,j} + 1 \leq b$.

On the other hand, if the sum of the digits of $\mathbf{w}_i$ equals b then $inc_1 = 0$ and $reset_1 = 1$. Denote by j^* the index of the first digit that carries a nonzero value. That is, $w_{i,j} = 0$ for all $1 \leq j < j^*$ and $w_{i,j^*} \geq 1$. From line 12, all the first $j^* - 1$ digits that carry a zero have $reset_j = 1$ and hence will remain zero (line 8). In addition, w_{i+1,j^*} will be set to zero since $reset_{j^*} = 1$, and the following digit that will have $reset_{j^*+1} = 0$ and $inc_{j^*+1} = 1$ will be incremented by one (line 10). Therefore we have

$$w_{i+1,j^*} + w_{i+1,j^*+1} = 0 + (1 + w_{i,j^*+1}) \leq w_{i,j^*} + w_{i,j^*+1}.$$

Since $reset_{j^*+1} = 0$ all the following digits, $j > j * +1$, will have $reset_j = inc_j = 0$. Hence, they will keep their value (line 9). Therefore,

$$\begin{aligned} \sum_{j=1}^{t} w_{i+1,j} &= 0 + w_{i+1,j^*} + w_{i+1,j^*+1} + \sum_{j=j^*+2}^{t} w_{i+1,j} \\ &\leq \sum_{j=j^*}^{t} w_{i,j} \leq \sum_{j=1}^{t} w_{i,j} = b \end{aligned} \tag{5}$$

and thus $\mathbf{w}_{i+1} \in \Omega_b$.

Next, we show that all the elements of Ω_b are generated. For this, it is sufficient to show that

$$N(\mathbf{w}_i) < N(\mathbf{w}_{i+1}) \tag{6}$$

and that there is no legal vector, say $\hat{\mathbf{w}}$, in Ω_b such that $N(\mathbf{w}_i) < N(\hat{\mathbf{w}}) < N(\mathbf{w}_{i+1})$.

Note that if the sum of the digits of $\mathbf{w}_i$ is smaller than b, then $N(\mathbf{w}_{i+1}) = N(\mathbf{w}_i) + 1$ and Eq. 6 is fulfilled. Otherwise,

$$\begin{aligned} N(\mathbf{w}_{i+1}) - N(\mathbf{w}_i) &= q^{j^*}\left((w_{i+1,j^*} + q w_{i+1,j^*+1})\right) \\ &\quad - q^{j^*}\left((w_{i,j^*} + q w_{i,j^*+1})\right) \\ &= q^{j^*}(q - w_{i,j^*}) > 0. \end{aligned} \tag{7}$$

Assume now that there exists a legal vector $\hat{\mathbf{w}}$ in between $\mathbf{w}_{i+1}$ and $\mathbf{w}_i$. Then, $\hat{w}_{j^*+1} \in \{w_{i,j^*+1}, w_{i+1,j^*+1}\}$. If $\hat{w}_{j^*+1} = w_{i,j^*+1}$ then $\sum_{j=1}^{t} \hat{w}_j > \sum_{j=1}^{t} w_{i,j} = b$ is hence a contradiction (since $\hat{\mathbf{w}}$ cannot be a member of Ω_b). If $\hat{w}_{j^*+1} = w_{i+1,j^*+1}$ then $\sum_{j=1}^{j^*} \hat{w}_j < \sum_{j=1}^{j^*} w_{i+1,j} = 0$ and this again is a contradiction since the sum of the first j^* digits of $\hat{\mathbf{w}}$ cannot be negative. □

B. Proof of Theorem 3

At the first time slot the registers R_{j^+} are initialized to carry the value 1. (In Section V-F, there are four sets of registers, where each set is initialized with a different predefined value.)

Consider the i'th time slot, $i > 1$. Assume that the last time slot R_{j^+} was updated was at time p, $p < i$. At that time slot, the increment occurred at position $j_p^+ \geq j_i^+$ and $\mathbf{w}_p$ carried the value

$$\mathbf{w}_p = \begin{cases} (w_{p,t}, \ldots, w_{p,j_p^+}, 0, \ldots 0, \underbrace{0}_{j_i^+}, 0, \ldots 0) & for \quad j_p^+ > j_i^+, \\ (w_{p,t}, \ldots, w_{p,j_p^+}, 0, 0, \ldots 0) & for \quad j_p^+ = j_i + \end{cases}.$$

That is, at time slot i the register $R_{j_i^+}$ holds $X^{\mathbf{w}_p}$. In the time slots between p and i, the value of the $\mathbf{w}$ vectors were in the range

$$N(\mathbf{w}_p) < N(\Omega) < N(\mathbf{w}_p) + q^{j_i^+}$$

because the counter generates an increasing series of numbers; namely, all the registers with indices smaller than j_i^+ were changed, and the registers with indices greater or equal than j_i^+ remained untouched. In other words, at time slot i, the $\mathbf{w}_i$ vector is of the form

$$\begin{aligned} \mathbf{w}_i &= (w_{i-1,t}, \ldots, w_{i-1,j_i^+ + 1}, w_{i-1,j_i^+} + 1, 0, \ldots 0) \\ &= \mathbf{w}_p + (0, \ldots, 0, 1, 0 \ldots 0). \end{aligned} \tag{8}$$

Hence, $P_i = x^{w_{j_i^+}} X^{\mathbf{w}_p} = X^{\mathbf{w}_i}$ and the block outputs the correct value. □

C. Examples

In the following examples the computations were run over the finite field $\mathbb{F}_2^8$. We used the primitive polynomial $\pi(x) = x^8 + x^4 + x^3 + x^2 + 1$ to construct the field.

Example 8 (The error-free case)**.** *Consider the following consecutive basic block complied for an ARM architecture.*

```
add.w r2, r7, #16
subs r2, #8
bl 558          //end of BB1
```

```
movs r1, #1
bl 0   //end of BB2
```

The binary of BB1 is the following (3×32)*-bit vector*

$$\begin{aligned} Y = (y_{12}, \ldots, y_1) = (\ & \texttt{0xEB}, \texttt{0x00}, \texttt{0x00}, \texttt{0x87}, \\ & \texttt{0xE2}, \texttt{0x52}, \texttt{0x20}, \texttt{0x08}, \\ & \texttt{0xE2}, \texttt{0x87}, \texttt{0x20}, \texttt{0x10} \ ). \end{aligned}$$

Assume that the randomly chosen part attached to Y *is*

$$X = (x_3, x_2, x_1) = (\texttt{0x87}, \texttt{0x37}, \texttt{0x1C}),$$

then the pre-computed tag is $f(X, Y) = \texttt{0x3C}$*. In an* ***error-free scenario****, the checker recomputes the tag and sees that it equals the tag stored in the tamper-resistant memory. In this case, the checker works as follows:*

When the previous basic block ends and BB1 begins the counters and registers are initialized to:

$$\begin{aligned} &(2,9)\text{-}cnt^{(0)} = (0,1), \quad R_1^{(0)} = 1, \quad R_2^{(0)} = 1, \\ &(2,8)\text{-}cnt^{(1)} = (0,0), \quad R_1^{(1)} = x_3, \quad R_2^{(1)} = x_3, \\ &(2,7)\text{-}cnt^{(2)} = (0,0), \quad R_1^{(2)} = x_3^2, \quad R_2^{(2)} = x_3^2, \\ &(2,6)\text{-}cnt^{(3)} = (0,0), \quad R_1^{(3)} = x_3^3, \quad R_2^{(3)} = x_3^3. \end{aligned}$$

The first fetched instruction $(\texttt{0xE2}, \texttt{0x87}, \texttt{0x20}, \texttt{0x10})$ *is bought to the checker, and the four bytes are split between the four parallel units. The values at the output of the Polynomial Evaluation module are*

$$\begin{aligned} f_{1,0}(X,Y) &= y_1 x_1 = x^4 \cdot (x^4 + x^3 + x^2) \mod \pi(x) = \texttt{0xDD} \\ f_{1,1}(X,Y) &= y_2 x_3 = \texttt{0x5A} \\ f_{1,2}(X,Y) &= y_3 y_3^2 = \texttt{0x35} \\ f_{1,3}(X,Y) &= y_4 x_3^3 = \texttt{0xDA} \\ f_1(X,Y) &= \bigoplus_{i=0}^{3} f_{1,i} = \texttt{0x68} \end{aligned}$$

The tag value at the end of the first cycle is $\texttt{0x68}$*.*

At the beginning of the second cycle the values of the counters and registers are

$$\begin{array}{lll} (2,9)\text{-}cnt^{(0)} = (0,2), & R_1^{(0)} = x_1, & R_2^{(0)} = 1, \\ (2,8)\text{-}cnt^{(1)} = (0,1), & R_1^{(1)} = x_3, & R_2^{(1)} = x_3, \\ (2,7)\text{-}cnt^{(2)} = (0,1), & R_1^{(2)} = x_3^2, & R_2^{(2)} = x_3^2, \\ (2,6)\text{-}cnt^{(3)} = (0,1), & R_1^{(3)} = x_3^3, & R_2^{(3)} = x_3^3. \end{array}$$

In this cycle, the second instruction, $(\texttt{0xE2}, \texttt{0x52}, \texttt{0x20}, \texttt{0x08})$*, is fetched and the checker computes*

$$\begin{array}{rcl} f_{2,0}(X,Y) & = & f_{1,0} \oplus y_5 x_1^2 = \texttt{0x8F} \\ f_{2,1}(X,Y) & = & f_{1,1} \oplus y_6 x_3 x_1 = \texttt{0x71} \\ f_{2,2}(X,Y) & = & f_{1,2} \oplus y_7 x_3^2 x_1 = \texttt{0xA6} \\ f_{2,3}(X,Y) & = & f_{1,3} \oplus y_8 x_3^3 x_1 = \texttt{0x2A} \\ f_2(X,Y) & = & \bigoplus_{i=0}^{3} f_{2,i} = \texttt{0x72}. \end{array}$$

That is, the value of the tag at the end of the second cycle; i.e., the tag accumulated so far, is 0x72*.*

Finally, in the third cycle, the last instruction of the basic block, $(\texttt{0xEB}, \texttt{0x00}, \texttt{0x00}, \texttt{0x87})$ *is bought to the checker;*

the values of the counters and registers are

$$\begin{array}{lll} (2,9)\text{-}cnt^{(0)} = (0,3), & R_1^{(0)} = x_1^2, & R_2^{(0)} = 1, \\ (2,8)\text{-}cnt^{(1)} = (0,2), & R_1^{(1)} = x_3 x_1, & R_2^{(1)} = x_3, \\ (2,7)\text{-}cnt^{(2)} = (0,2), & R_1^{(2)} = x_3^2 x_1, & R_2^{(2)} = x_3^2, \\ (2,6)\text{-}cnt^{(3)} = (0,2), & R_1^{(3)} = x_3^3 x_1, & R_2^{(3)} = x_3^3. \end{array}$$

The checker computes

$$\begin{array}{rcl} f_{3,0}(X,Y) & = & f_{2,0} \oplus y_9 x_1^3 = \texttt{0x07} \\ f_{3,1}(X,Y) & = & f_{2,1} \oplus y_{10} x_3 x_1^2 = \texttt{0x71} \\ f_{3,2}(X,Y) & = & f_{2,2} \oplus y_{11} x_3^2 x_1^2 = \texttt{0xA6} \\ f_{3,3}(X,Y) & = & f_{2,3} \oplus y_{12} x_3^3 x_1^2 = \texttt{0xEC} \\ f_3(X,Y) & = & \texttt{0x3C} \end{array}$$

After the third cycle the block ends; as expected, the computed tag 0x3C *equals the pre-computed one.*

□

Example 9 (An adversary tampering with the content of a basic block)**.** *Assume that the BB1 from Example 8 has been altered by an adversary that has injected bit-flips to obtain the following code (the tampered-with parts are written in bold):*

add.w r2, ***r1****, #16*
add.w r3, r4, #1
bl 558 //end of BB1

---

movs r1, #1
bl 0 //end of BB2

The corresponding binary vector is

$$\hat{Y} = (\hat{Y}_{12}, \ldots, \hat{Y}_1) = (\begin{array}{l} \texttt{0xEB}, \texttt{0x00}, \texttt{0x00}, \texttt{0x87}, \\ \texttt{0xE2}, \texttt{0x}\boldsymbol{84}, \texttt{0x}\boldsymbol{3}\texttt{0}, \texttt{0x0}\boldsymbol{1}, \\ \texttt{0xE2}, \texttt{0x8}\boldsymbol{1}, \texttt{0x20}, \texttt{0x10} \end{array})$$

(the differences between Y *and* $\hat{Y}$ *are marked in bold).*

The tag computation process is similar to the computation in Example 8. After the third instruction the computed tag which equals $f(X,\hat{Y}) = \texttt{0xB2}$ *is compared with the tag stored in memory (*0x3C*). Since the tags differ, the checker signals that an error has been detected.*

It is important to note that in this case the actual error masking probability $Q(E)$ *is smaller than the error masking probability of the code* $\bar{Q} = 8 \cdot 256^2/(256^3 - 1)$*; since*

$$\max\{\deg(f(X,Y), \deg(f(X,\hat{Y})\} = 3,$$

at least $256^2(256-3)$ X*'s out of* 256^3 *are able to detect this code manipulation. In other words, the error masking probability for this specific error vector,*

$$E = (\begin{array}{l} \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \\ \texttt{0x00}, \texttt{0xD6}, \texttt{0x10}, \texttt{0x09}, \\ \texttt{0x00}, \texttt{0x06}, \texttt{0x00}, \texttt{0x00} \end{array}),$$

is $Q(E) \leq \frac{3}{256-1} < \bar{Q}$*.*

□

Example 10 (An adversary shortening a basic block)**.** *Consider BB1 from Example 8. By injecting bit-flips it is possible to change the second instruction (*subs*) into a branch instruction. This way the adversary shortens the basic block by one instruction:*

add.w r2, r7, #16
bl 100 //end the modified BB1

---

bl 558 //end the original BB1
\- -
movs r1, #1
bl 0 //end of BB2

Now the checker will "see" a shorter vector

$$\hat{Y} = (\hat{y}_8, \ldots, \hat{y}_1) = (\begin{array}{l} \texttt{0xEB}, \texttt{0x00}, \texttt{0x00}, \texttt{0x16}, \\ \texttt{0xE2}, \texttt{0x87}, \texttt{0x20}, \texttt{0x10} \end{array})$$

and will check the signature at the end of the second cycle. The computed signature is

$$f_1(X,\hat{Y}) = \hat{y}_1 x_1 \oplus \hat{y}_2 x_3 \oplus y_3 x_3^2 \oplus y_4 x_3^3 = \texttt{0x68}$$

$$f_2(X,\hat{Y}) = f_1(X,\hat{Y}) \oplus \hat{y}_5 x_1^2 \oplus \hat{y}_6 x_3 x_1 \oplus \hat{y}_7 x_3^2 x_1 \oplus \hat{y}_8 x_3^3 x_1 = \texttt{0x7D}$$

Hence, $f(X,\hat{Y}) = \texttt{0x7D} \neq f(X,Y)$*, and the error is detected.*

Similar to Example 9, $\max\{\deg(f(X,Y), \deg(f(X,\hat{Y})\} = 3$*. Consequently, at least* $256^2(256-3)$ X*'s out of* 256^3 *are able to detect this code manipulation. In other words, the error masking probability for this specific error vector,*

$$E = (\begin{array}{l} \texttt{0xEB}, \texttt{0x00}, \texttt{0x00}, \texttt{0x87}, \\ \texttt{0x00}, \texttt{0x52}, \texttt{0x20}, \texttt{0x1E}, \\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \end{array}),$$

is $Q(E) \leq \frac{3}{256-1} < \bar{Q}$*.* □

Example 11 (An adversary lengthening a basic block)**.** *Consider BB1 from Example 8. Assume that an adversary has lengthened this basic block by changing the branch instruction into a non-branch instruction. For example, assume that the tampered-with code becomes*

add.w r2, r7, #16
subs r2, #8
add.w r1, r1, #2 //end of the original BB1
\- -
bl 100 //end of the modified BB1

---

bl 0 //end of BB2

The checker will "see" the vector

$$\hat{Y} = (\hat{y}_{16}, \ldots, \hat{y}_1) = (\quad \texttt{0xEB}, \texttt{0x00}, \texttt{0x00}, \texttt{0x14}, \\ \texttt{0xE2}, \texttt{0x81}, \texttt{0x10}, \texttt{0x02}, \\ \texttt{0xE2}, \texttt{0x52}, \texttt{0x20}, \texttt{0x08}, \\ \texttt{0xE2}, \texttt{0x87}, \texttt{0x20}, \texttt{0x10} \quad).$$

The checker will compare the computed tag value after the fourth cycle. Since the first two instructions were not changed, the computed tag value in the first and second cycles are identical to the first two tags in Example 8. In other words, we have,

$$f_1(X, \hat{Y}) = \hat{y}_1 x_1 \oplus \hat{y}_2 x_3 \oplus \hat{y}_3 x_3^2 \oplus \hat{y}_4 x_3^3 = \texttt{0x68}$$

$$f_2(X, \hat{Y}) = f_1(X, \hat{Y}) \oplus \hat{y}_5 x_1^2 \oplus \hat{y}_6 x_3 x_1 \oplus \hat{y}_7 x_3^2 x_1 \oplus \hat{y}_8 x_3^3 x_1 = \texttt{0x72}$$

$$f_3(X, \hat{Y}) = f_2(X, \hat{Y}) \oplus \hat{y}_9 x_1^3 \oplus \hat{y}_{10} x_3 x_1^2 \oplus \hat{y}_{11} x_3^2 x_1^2 \oplus \hat{y}_{12} x_3^3 x_1^2 = \texttt{0xBA}$$

$$f_4(X, \hat{Y}) = f_3(X, \hat{Y}) \oplus \hat{y}_{13} x_1^4 \oplus \hat{y}_{14} x_3 x_1^3 \oplus \hat{y}_{15} x_3^2 x_1^3 \oplus \hat{y}_{16} x_3^3 x_1^3 = \texttt{0x3B}$$

Since the computed tag $f(X, \hat{Y}) = f_4(X, \hat{Y}) = \texttt{0x3B}$ differs from the tag read from the tamper-resistant memory, the error will be detected.

Note that in this case $256^3 - 4 \cdot 256^2$ X's out of 256^3 are able to detect this code manipulation. In other words, the error masking probability for this error

$$E = (\quad \texttt{0xEB}, \texttt{0x00}, \texttt{0x00}, \texttt{0x14}, \\ \texttt{0x09}, \texttt{0x81}, \texttt{0x10}, \texttt{0x85}, \\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \quad).$$

is $Q(E) \leq \frac{4}{256} < \bar{Q}$.
□

Example 12 (The weakness of linear codes)**.** *This example shows that a linear code; e.g., the code in [11], will always have an error masking probability $\bar{Q} = 1$. Furthermore, even the use of a random vector X for masking the signature cannot solve the problem.*

Assume that a basic block is protected by a CRC32 code with the generator polynomial $g(z)$ whose coefficients are $G = \texttt{0x82F63B78}$ and a random mask X. Formally, let R_Y be the coefficients of the reminder polynomial

$$r_Y(z) = Y(z)z^{32} \mod g(z),$$

then the tag function is $f(Y) = R_Y \oplus X$.

Consider the basic block from Example 8, and let the (secret) random mask be $X = \texttt{0xFFFFFFFF}$. The corresponding tag, $f(Y) = \texttt{0x47718DEF} \oplus X = \texttt{0xB88E7210}$, is stored in a tamper-resistant memory. Assume that an adversary with the ability to inject precise errors injects the error vector

$$E = (\quad \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x07}, \\ \texttt{0xB7}, \texttt{0x1B}, \texttt{0xD0}, \texttt{0x50}, \quad . \\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \quad).$$

Then, the corresponding tampered code will be

---

*add.w r2, **r1**, #16*
strbpl pc, [sb, #-0x58]
*bl **528** //end of BB1*

---

movs r1, #1
bl 0 //end of BB2

For this error vector we have $r_Y(z) = r_{\hat{Y}}(z)$ for every random mask E. Since the calculated tag of the modified basic block, $f(\hat{Y})$, is identical to one stored in the memory, the error will never be detected ($Q(E) = 1$). In fact, any error vector E which is a multiple of the generator polynomial, will be masked. □

Example 13 (Error-recovery mechanism can help the adversary)**.** *Throughout this paper, we assumed that the adversary has the ability to choose the error. In practice, it is hard to inject precise errors, and thus, a fault can yield a non-valid opcode . As we show next, a decoder with error correction capabilities that attempts to recover from this error and correct the opcode may cause security issues.*

Consider the basic block, Y, and the corresponding CRC32 tag, $f(Y)$, from Example 12. Assume that the adversary failed to inject the intended error E, and instead flipped the bits such that the actual error is

$$E' = (\quad \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x07}, \\ \texttt{0xB}\boldsymbol{\texttt{1}}, \texttt{0x1B}, \texttt{0xD0}, \texttt{0x50}, \quad . \\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00} \quad).$$

(the differences between E and E' are marked in bold).

In this case, the decoder finds out that the second instruction in the erroneous basic block, $\hat{Y}$, yields an invalid opcode. Consequently, the tuple $\hat{c} = (Y + E', f(Y))$ is not a valid codeword and the decoder activates the error correction mechanism. It decodes the erroneous word $\hat{c}$ into a codeword $c' \in \mathcal{C}$ for which the Hamming distance $d(c', \hat{c})$ is minimal. From the properties of the CRC32 code it follows that c' is the codeword $(Y + E, f(Y))$ from Example 12. In other words, rather than correcting the basic block, the decoder helped the attacker to conduct a successful attack. □