Nonlinear Codes for Control Flow Checking

Giorgio Di Natale, O. Keren

Nonlinear Codes for Control Flow Checking

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 to detect both errors in the control flow and the instruction stream being executed on a processor. However, as was shown in recent papers, these techniques fail to detect malicious carefully-tuned manipulation of the instruction stream in a basic block. This paper presents a non-linear encoder and checker that can cope with this weakness.**

I. Introduction

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

Many techniques have been proposed in literature to cope with transient, permanent and malicious faults in many parts of a system. They target both the hardware and the software parts, and rely on different forms of redundancy. Among these techniques for reliability improvement and fault tolerance, Control Flow Checking (CFC) allows covering faults affecting storing elements containing the executable program, as well as all the hardware components handling the program itself and its flow.

CFC has been proposed to cope with reliability issues for both transient and permanent faults ([1], [2]) and more recently, with security issues caused by the injection of malicious faults [3], [4], which can allow an attacker to either bypass security checks or retrieve secret information. Software based CFC solutions which modify the code rely on the assumption that the code stored in memory is not being maliciously tampered with and thus cannot provide security [5]; on the other hand, 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]. Fine grained CFC policy allows control flow along valid edges of the control flow graph whereas coarse grain policy relaxes this restriction. A control flow graph (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. The work in [7] 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. The fine-grained methods include Shadow Call Stack protection which is designed to detect tampering with return addresses stored in the stack during a function call [7, 8], Code Pointer Integrity which uses short Message Authentication Code (MAC) tags to verify the authentication of pointers at run time [9], and the Signature Modeling technique [10, 11, 12].

In Signature Modeling, basic blocks are accompanied by a signature, such as a Cyclic Redundancy Check (CRC) checksum or Hamming code, generated at run-time and then compared against a pre-computed signature which is stored in a secure memory. In 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 [11, 12] or when a basic block ends [7, 3]. In [12] a CRC-based signature monitor was integrated into the instruction fetch state to prevent the processing of instructions whose pre-calculated and the current signatures do not match. The authors in [13] proposed a technique to map a malicious software into another one (protected by a control flow checking mechanism), without violating the structure of the latter one, in other words, 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 MAC, the signature is calculated by resorting also to a secret information which, besides data integrity, allows guaranteeing the authenticity of the BB.

In this paper we propose a signature calculation based on non-linear codes, to protect against malicious modifications of the executed program. The proposed method can be applied to fine-grained CFC schemes. We assume the attacker knows the protected architecture details and its machine language, as well as the program and its control flow graph. Moreover, the attacker has the means to execute malicious physical manipulations on the device by injecting precise faults at run-time into the machine code stored in memory. Our contribution is the following:

- A non-linear code based on a weakened version of the multiple random variables Karpovsky-Wang Algebraic Manipulation Detection (AMD) code [14]
- A signature calculation method that works in parallel to the processor pipeline and does not require processor changes, code changes or additional latency.
- By making use of the fact that the signature is stored in a secure memory and cannot be tampered with, the area overhead of the signature calculation is relatively

*Institute of Engineering Univ. Grenoble Alpes

small (compared to the methods able to prevent malicious attacks) and does not require partitioning the program into basic blocks of equal length as required in [7].

This paper is organized as follows: Section 2 presents an overview of existing CFC solutions, as well it details the architecture in which the proposed signature calculation can be used. Section 3 presents some definitions and the security metric we use to evaluate and build the non-linear code. Section 4 presents the theoretical construction of the code, while Section 5 describes its hardware implementation. Finally, we draw some conclusions in section 6.

II. Context

Error-detection codes used for integrity checking in CFC schemes were historically developed to have an overall small impact on the target system, in terms of area overhead and additional delay introduced for its calculation. These solutions were targeting primarily natural faults, which have a uniform statistical distribution. However, in order to cope also with malicious attacks, new methods have been proposed, based on Message Authentication Codes (MAC). In MAC, the signature is calculated by resorting also to a secret information which, besides data integrity, allows guaranteeing the authenticity of the data.

Nevertheless, the MAC techniques that are based on statically computed cryptographic hash of the instruction sequence in the basic block [7, 15] have generally 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. Few MAC based checkers ([6, 12]) allows the computation of the signature together with the execution of the program itself. Nevertheless, these solutions have some 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. 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 [12], on the other hand, 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 detect only a (relatively) small number of errors, and they cannot detect attacks made by sophisticated precise attackers.

In this paper we present a MAC based Control Flow Checker that has the advantage of working with basic blocks of variable length, detecting any injected error and performing the computation without execution time overhead.

The proposed MAC scheme can be applied to any CFC architecture where the co-processor in charge of calculating the signatures of the BBs (also called control flow checker, or *watchdog*) works in parallel with the main processor, by fetching the instructions to be executed from the main bus, and by comparing the obtained signature with a predefined one. Fig. 1 shows the generic architecture in which the MAC can be used. The watchdog is a standalone module that works in parallel to processor's pipeline. It does not modify the pipeline stages, does not add latency, nor interfere with the program flow. The CFC communicates with the processor via the existing interface; it receives its inputs which are the current address and instruction on buses used by the processor during the Instruction Fetch (IF) phase. The CFC is implemented in a secure zone. As shown in Fig. 1, the control flow checker consists of four main blocks: a compact processing unit, a comparator, a control unit, and a secure memory array (e.g., the secure RAM presented in [16]). The secure memory is more expensive, but is used for storing only a small amount of information and not the whole program. The size of the secure memory and its width depends on the number of basic blocks, their maximal size and the required security level.

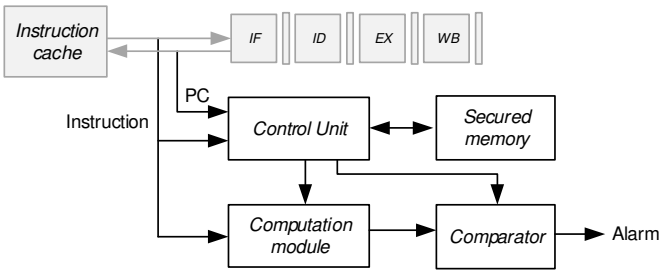

Fig. 1: Micro-architecture of the control flow checker

We assume that:

- The off-line signature calculation process as well as the program can be trusted.
- To reduce the cost of the product, the main memory has no dedicated security protection whereas the watchdog itself, including its secure memory in which the signatures are stored, is not accessible by 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 the cache and is able to inject arbitrary or precise errors at run-time; i.e., when the code is loaded from the memory into the cache or when it is being fetched from the cache.

III. Definitions and Security metric

A Basic Block (BB) is a piece of code made of one or several consecutive instructions without any jumps between them. A BB starts when its address is the target of a jump instruction of another (or others) BB(s), and ends with a jump to another BB (or a return). The BB size is the number of bytes (k) occupied by all the instructions in that BB.

Most of the security oriented codes[1] are defined over a finite field $\mathbb{F}_q$ of size q. The size of the field determines the complexity of the arithmetic over that field. In this paper we work over $\mathbb{F}_q$, $q = 2^r$. As we show in the next section, r determines both the effectiveness of the code and the implementation

[1]Most of the codes are defined over finite fields, but there are codes that are defined over integer rings. See for example [17, 18, 19].

complexity. The smaller r is, the lower implementation cost of the multipliers over that field.

The sequence of instructions in a BB is modeled as a binary vector. Because of the limits of the code (as explain in Section IV), we assume each BB to have a size smaller or equal to $8N$ bits (i.e., N bytes). The encoder that computes the signature and the checker that verifies the validity of the sequence refer to this sequence as a q-ary vector of length k

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

where

$$k \leq k_{max} = \lceil \frac{8 \cdot N}{r} \rceil.$$

Each sequence Y is associated with a signature S. The signature is computed off-line and stored in a secured memory. The pair (Y, S) can be referred to as a codeword of a variable length (security oriented) code. Each signature S has two parts, a random part X (i.e., the secret information) and a computed part $f(X, Y)$. It is assumed that the attacker knows Y but does not know X.

The signature itself is a binary vector of length $(t+1)r$ bits. (As we show in next section, larger t provides better security). We refer to S as a q-ary vector of length $t+1$. Its random part is a **nonzero** q-ary random vector of length t,

$$X = (x_t, ...x_1) \in \mathbb{F}_q^t,$$

and $f = f(Y, X) \in \mathbb{F}_q$ is a **single** q-ary symbol.

Denote by $\hat{Y} = (\hat{y}_{\hat{k}}, ...\hat{y}_1)$ the (possibly distorted) sequence read by the checker. It is assumed that the attacker knows the original sequence Y, and can alter it as s/he wishes. Moreover, the attacker may also change the length of the sequence (i.e., $k \neq \hat{k}$). The signature is not observable and hence cannot be altered. Thereby, the checker "sees" the tuple $(\hat{Y}, X, f(X, Y))$.

Our goal is to construct an *online* control flow checker that receives the content of the BB as read from the memory $(\hat{Y})$ and the address of the first instruction of the next BB (i.e., it knows i), and computes in parallel to the execution of the BB the value of $f(X_i, \hat{Y})$. The checker rises a flag if the computed value differs from the one stored in the secured memory.

The effectiveness of this control flow checker is defined as the probability Q that the worst attack will pass unnoticed. That is, let X be a uniformly distributed vector over a subset $\mathcal{X} \subseteq \mathbb{F}_q^t$, then

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

where $E_X()$ means the expected value with respect to X.

IV. Construction

The code presented in this paper is a derivative of Karpovsky-Wang code [14]. However, its computational complexity is smaller since it makes use of the fact that the signature cannot be tampered with. This property enables us to construct a variable length code whose checker can work in parallel to the execution of the BB without changing its throughput or latency. It also enables a simple and smooth transition between BBs.

The following coding scheme is based on the Generalized Reed-Muller (GRM) codes [20]. It has three parameters, r, t and b. r defines that size of the field ($q = 2^r$), t is the number of random q-ary symbols, and $b < q-1$ is the smallest integer for which the coding scheme can protect a sequence of maximal length k_{max}. The value of b is determined as follows.

Let $\mathbb{Z}_q$ be the set of integers $\{0, 1, ..., q-1\}$. Let Ω be an ordered set of integer vectors whose sum is smaller of equal to b. That is,

$$\Omega_b = \{\Omega = (w_t, ...w_1) \ : \ 0 < \sum_{i=1}^{t} w_i \leq b, \text{ and } w_i \in \mathbb{Z}_q\}.$$

The parameter b is the smallest integer for which the size of Ω_b is equal to or greater than k_{max}. Namely,

$$|\Omega_b| = \binom{t+b}{b} - 1 \geq k_{max}.$$

We define a product term X^Ω as

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

where computation are performed over $\mathbb{F}_q$.

Construction 1. *Let Y be a q-ary vector of length $k \leq k_{max}$. A polynomial based non-linear signature of Y is a binary vector of size $(t+1) \cdot r$ bits of the form $S = (X, f)$ where*

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

and $\Omega_i \in \Omega_b$

In fact, $f(X, Y)$ is the X'th symbol in a GRM codeword c that is associated with the information symbol $(\mathbf{0}_{k_{max}-k}, Y)$. A schematic description of a control flow encoder and checker for $t = 1$ is given in Figure 2.

Example 1. *Let the maximal length of a sequence be $N = 1547$ bytes. Assume we want to design a control flow checker whose signature is a binary vector of length $30 = (2+1) \cdot 10$, that is, $r = 10$ and $t = 2$. Then we have $k_{max} = \lceil \frac{8 \cdot 1547}{10} \rceil = 1238$, and $b = 49$ is the smallest integer for which*

$$\binom{2+b}{b} - 1 = 1274 > k_{max}.$$

A signature is a binary vector of the form $S = (X = (x_1, x_2), f(Y, X))$ where x_1, x_2 and f are 10 bit vectors that represent elements from the finite field $\mathbb{F}_{2^{10}}$. The value of f for $k = k_{max}$ is computed as follows

$$\begin{aligned} f(Y, X) &= y_1 X^{(0,1)} + y_2 X^{(0,2)} + ...y_{49} X^{(0,49)} + \\ &\quad y_{50} X^{(1,0)} + y_{51} X^{(1,1)} + ...y_{98} X^{(1,48)} + \\ &\quad \vdots \\ &\quad y_{1230} X^{(41,0)} + y_{1231} X^{(41,1)} + ...y_{1238} X^{(41,8)} \\ &= y_1 x_1 + y_2 x_1^2 + ...y_{49} x_1^{49} + \\ &\quad y_{50} x_2 + y_{51} x_2 x_1 + ...y_{98} x_2 x_1^{48} + \\ &\quad \vdots \\ &\quad y_{1230} x_2^{41} + y_{1231} x_2^{41} x_1 + ...y_{1238} x_2^{41} x_1^8. \end{aligned}$$

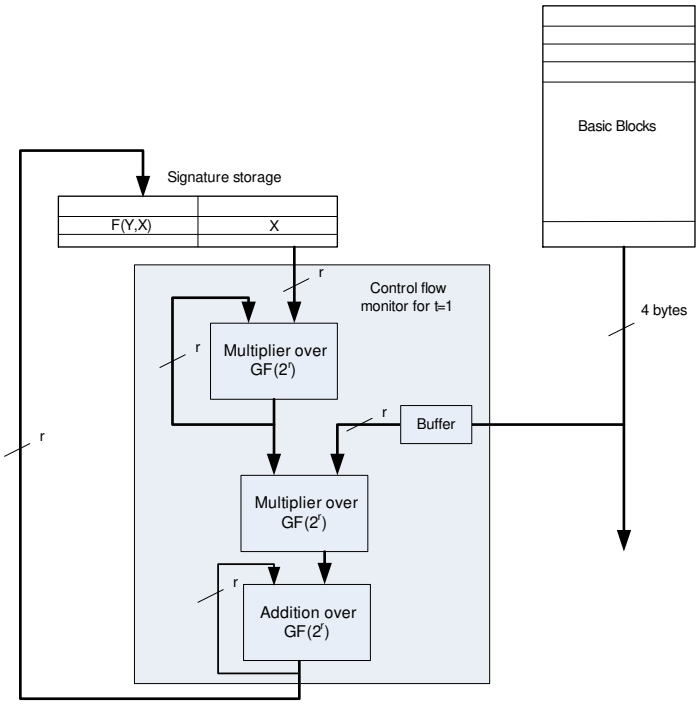

Fig. 2: A control flow encoder that computes $f(Y, X)$ for $t = 1$. The value of X is chosen at random during the computation of the signature. The buffer separates the sequence of instructions and the address of the next basic block into r-bit tuples.

As we show next, the probability that an attack will be masked is approximately $49/1024$.

Another way to construct a control flow checker for $N = 1547$ *bytes is by taking a larger* r*, i.e.,* $r = 16$ $t = 1$ *and* $b = \lceil N/2 \rceil = 774$*. 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,* $f(Y, x) = y_1 x + y_2 x^2 + \cdots + y_{774} x^{774}$*. Here, the computation is performed in the (larger) field* $\mathbb{F}_{2^{16}}$*, hence, the implementation cost is larger, however probability that an attack will be masked becomes smaller (* $774/2^{16}$*).*

A. Implementation considerations

The control flow checker computes the expected f in parallel to the execution of the BB. That is, the bytes read from the memory enter a buffer that groups them into r-tuples and sends each tuple (i.e. each q-ary symbol) to the checker. The checker circuit consists of a t digit counter that outputs Ω_i. The counter and its control signals are described in Alg. 1 lines 9-30. The counter is a radix $(b + 1)$ ripple counter. For example, for $t = 4, b = 15$, and $\Omega_i = (w_4 = 1, w_3 = 7, w_2 = 0, w_1 = 3)$, the next Ω will be $\Omega_{i+1} = (1, 7, 0, 4)$. Similarly, $\Omega_i = (0, 0, 0, 15)$ will be followed by $\Omega_{i+1} = (0, 0, 1, 0)$. Note that the counter described in Alg. 1 differs from the conventional t-digit counter. A conventional counter has a global reset/preset signal that initializes (simultaneously) all the digits to a predefined value, whereas our counter has t local reset and increment signals. That is, the counter can increment its upper part, e.g., $(w_t, \ldots, w_{j+1})$, and reset its lower part $(w_j, \ldots, w_1)$. For example, $\Omega_i = (4, 2, 8, 1)$ will be followed by

$$\Omega_{i+1} = (4, 2, 9, 0),\ \Omega_{i+2} = (4, 3, 0, 0), \Omega_{i+3} = (4, 3, 0, 1), \ldots.$$

The vector Ω_i that the counter outputs is used to compute $v = X^{\Omega_i}$ for the next symbol to arrive (Alg. 1, line 31). When a new symbol y arrives from the buffer, it is multiplied by v and the product is added to the computed f (Alg. 1, line 7). Notice that in each round only a **single** w, say w_{j^*}, is incremented. In fact, the remaining w_j's either keep their previous values (when $j > j^*$), or they are assigned with zeros (for $j < j^*$). Hence, it is not necessary to compute $x_j^{w_j}$ in each cycle (see line 31).

In order to simplify the implementation of this requirement, X has to take values from a subset $\mathcal{X} \subseteq \mathbb{F}_q^t \setminus \{0\}$. As we prove in Th. 1, the larger $\mathcal{X}$ is, the smaller the degradation in the code's effectiveness.

Algorithm 1 Checker algorithm

1: (Initialization step) Set $w_1 = 1$, $w_j := 0$ for all $1 = 2, \ldots, t$.
2: (Initialization step) Set $sum\Omega := 1$.
3: (Initialization step) Set $f := \mathbf{0}$.
4: (Initialization step) Read the X part of the signature, $X \in \mathcal{X}$.
5: (Initialization step) Set $v := x_1$. (v is initialized to x_1 because that w_1 was initialized to 1
6: **When a new y is read from the buffer,**
7: Compute $f := f + y \cdot v$
8: Compute the next v
9: **if** $sum\Omega == b$ **then**
10: Reset part of the lower digits and increment the upper part of the counter as follows:
11: **if** $w_1 == 0$ **then**
12: **if** $w_2 == 0$ **then**
13: $\vdots$
14: **if** $w_{t-2} == 0$ **then**
15: **if** $w_{t-1} = 0$ then $w_{t-1} := 0$ **end if**
16: Increment w_t.
17: **else**
18: Reset $(w_{t-2}, \ldots w_1)$ and increment (w_t, w_{t-1}).
19: **end if**
20: $\vdots$
21: **else**
22: Reset (w_2, w_1) and increment $(w_t, \ldots w_3)$.
23: **end if**
24: **else**
25: Reset w_1 and increment $(w_t, \ldots w_2)$.
26: **end if**
27: **else**
28: Increment $(w_t, \ldots w_1)$.
29: **end if**
30: Compute the new sum, $sum\Omega = w_t + \cdots + w_1$, using a t-operand adder, see [21]
31: Prepare the v for the next symbol $v = x_1^{w_1} \cdot x_2^{w_2} \cdots x_t^{w_t}$.

B. The effectiveness of the construction

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 length, 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 $\mathbb{F}_q^{k_{max}}$ and treat E as an additive error vector.

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*

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

Proof. Tampering 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,

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

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,

$$c = (g_E(0), g_E(1), ..., g_E(q^t-1)).$$

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

$$\frac{q^t - d}{|\mathcal{X}|} = \frac{bq^{t-1}}{|\mathcal{X}|}.$$

□

Table I shows several construction for different block and signature sizes. The first column gives the length of the maximal sequence, N, (in byte), the probability Q that an attack will be masked with and without the restriction on the X's, is given in the second and third column, respectively. The signature size (in bits) is written in the forth column, and the GRM parameters, r, t and b are given in columns 5-7.

TABLE I: Code parameters for signature size ≤ 32 bits

N (bytes)	Q with restriction	Q without restriction	Signature size (bits)	r	t	b
153	0.0998	0.0938	30	6	4	6
140	0.0640	0.0625	28	7	3	8
161	0.0316	0.0313	32	8	3	8
127	0.0127	0.0127	30	10	2	13
367	0.1331	0.1250	30	6	4	8
315	0.0880	0.0859	28	7	3	11
282	0.0395	0.0391	32	8	3	10
258	0.0186	0.0186	30	10	2	19
532	0.1498	0.1406	30	6	4	9
591	0.1120	0.1094	28	7	3	14
525	0.1220	0.1211	24	8	2	31
556	0.0514	0.0508	32	8	3	13
519	0.0569	0.0566	27	9	2	29
540	0.0274	0.0273	30	10	2	28
1361	0.1997	0.1875	30	6	4	12
1136	0.0672	0.0664	32	8	3	17
1072	0.0391	0.0391	30	10	2	40

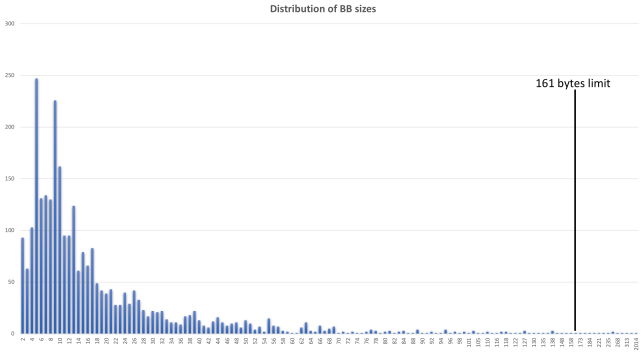

Fig. 3: Sizes of Basic Blocks taken from some real Linux-based applications

V. Implementation

The implementation cost (in terms of area overhead) depends on the choice of the code parameters. More in particular, the higher the t and b, the higher the overall cost. Moreover, the value of r will impact the overall architecture, since the size of the signature depends on it (i.e., it is equal to $t * r$). In order to work with modern 32-bit processors, we have selected $r = 8$. Among the possible combinations of $r = 8$, we hace checked the smallest values in Table I. More in particular, the third line shows the combination with $t = 3$ and $b = 8$, having a signature of 32 bits and a maximum BB size of 161 bytes. We have checked the sizes of BB on some actual applications, to understand the limitations of having $N = 161$. Figure 3 shows the distribution of BB's sizes, taken from some real Linux-based applications (we have merged the results from *ghostscript, head, hexdump, sort, tail*, running on a x86 architecture). As it can be seen, the vast majority of BBs have a number of bytes that is smaller than $N = 161$, thus confirming that the choice of these parameters is reasonable. It must be noticed that in case of bigger BBs, the original program will be modified in order to split the big BB into smaller BBs (by adding unconditional jump operations to link them).

The coding scheme presented in Section IV has been implemented for the chosen parameters. The implementation details are provided in Fig. 4. We have synthesized the circuit by using

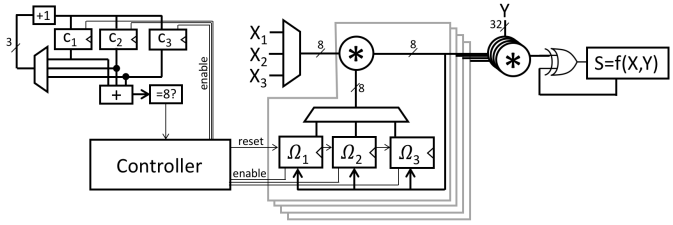

Fig. 4: Implementation details

TABLE II: Area Occupancy comparison

Solution	Area [GEs]
Proposed	1700
[6]	90K
[14]	N/A (requires multipliers of 36, 60, 90 bits)
[15]	N/A (requires an AES)
[18]	3500

a 90nm 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 the existing ones, we have calculated (when possible) the area of the other solutions in GEs. The values we obtained are sensitive to errors since not all technological details are provided (nor units in some cases). For instance, in [6], sizes are not explicitly calculated. However, they declare a 30% area overhead w.r.t. Leon3. Leon3 implementations vary from 300K to 450K GE, thus leading to a rough approximation of at least 90K GEs for their implementation. Table II presents the comparison with the other works discussed in this paper. As it can been seen, our solution has the smallest overhead, it does not introduce any latency, and it guarantees a high level of security.

VI. Conclusions

This paper presented a non-linear encoder and checker that can be used in any Control Flow Checking mechanism that uses a signature for every BB. It has the advantage of not introducing latency in its run-time calculation (similarly to existing solutions based on linear codes) while it guarantees high level of security of the MAC-based system, without introducing high area penalties. As for MAC-based solutions, the code integrates a secret part, which must be stored (together with the pre-calculated signatures) in a tamper-proof secure 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] S. Das, W. Zhang, and Y. Liu, "A fine-grained control flow integrity approach against runtime memory attacks for embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, pp. 3193–3207, Nov 2016.

[9] P. Qiu, Y. Lyu, J. Zhang, D. Wang, and G. Qu, "Control flow integrity based on lightweight encryption architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 1358–1369, July 2018.

[10] 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.

[11] 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.

[12] 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.

[13] G. Di Natale, M. L. Flottes, S. Dupuis, and B. Rouzeyre, "Hacking the control flow error detection mechanism," in *IEEE 2nd International Verification and Security Workshop (IVSW), Thessaloniki*, pp. 51–56, IEEE, 2017.

[14] 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.

[15] 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.

[16] 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.

[17] G. Gaubatz, B. Sunar, and M. G. Karpovsky, "Non-linear residue codes for robust public-key arithmetic," in *Fault Diagnosis and Tolerance in Cryptography*, pp. 173–184, Springer, 2006.

[18] 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.

[19] Y. Neumeier and O. Keren, "Expurgated codes for detecting jamming in multi-level memories," in *DEPEND 2016, The Ninth International Conference on Dependability*, pp. 15–20, iaria, 2016.

[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] I. Koren, "Computer arithmetic algorithms," in *Computer Arithmetic Algorithms*, A. K. Peters, CRC Press, 2002.