Instruction Flow-based Detectors against Fault Injection Attacks

Köylü, Troya Çağıl; Reinbrecht, Cezar; Brandalero, Marcelo; Hamdioui, Said; Taouil, Mottaqiallah

DOI
10.1016/j.micpro.2022.104638

Publication date
2022

Document Version
Accepted author manuscript

Published in
Microprocessors and Microsystems

Instruction Flow-based Detectors against Fault Injection Attacks

Troya Çağıl Köylü[a], Cezar Rodolfo Wedig Reinbrecht[a], Marcelo Brandalero[b], Said Hamdioui[a], Mottaqiallah Taouil[a]

[a]*Delft University of Technology, Delft, South Holland, the Netherlands*
[b]*Brandenburg University of Technology, Cottbus-Senftenberg, Brandenburg, Germany*

Abstract

Fault injection attacks are a threat to all digital systems, especially to the ones conducting security sensitive operations. Recently, the strategy of observing the instruction flow to detect attacks has gained popularity. In this paper, we provide a comparative study between three hardware-based techniques (i.e., recurrent neural network (RNN), content addressable memory (CAM), and Bloom filter (BF)) that detect fault attacks against software RSA decryption. After conducting experiments with various fault models, we observed that the CAM provides the best detection rate, the RNN provides the most software/application flexibility, and the BF is a middle ground between the two. Regardless, all of them exhibit robustness against faults targeted at them, and obtain a very high detection rate when faults change instructions altogether. This affirms the validity of monitoring the integrity of the instruction flow as a strong countermeasure against any type of fault attack.

Keywords: fault injection, countermeasure, machine learning, recurrent neural network, content addressable memory, Bloom filter

1. Introduction

Fault injection attacks are among the most important threats in current electronic systems [1]. Attackers can provoke hardware faults to modify functionality or steal sensitive data from devices such as personal computers, smartphones, and smartcards (e.g., bank, personal identification cards) [2, 3]. Modifying the functionality allows an adversary to take control of the system or bypass some security mechanism [4], while stealing data can happen by observing the output of faulty calculations during some cryptographic function [5]. This is an alarming issue, as many ways to inject faults have been proposed over the years. Examples include voltage underfeeding [6], voltage glitching [7], overclocking [8], clock glitching [2], heating [9], electromagnetic-based glitching [10], and laser glitching [11]. This paper addresses the issue of protecting asymmetric cryptography, the software RSA decryption in particular from these attacks. RSA is a very popular algorithm that is used in a variety of security sensitive applications, such as bank transactions, cloud authentication, and as key exchange protocols for symmetric cryptosystems [12, 13].

Email addresses: T.C.Koylu@tudelft.nl (Troya Çağıl Köylü), C.R.W.Reinbrecht@tudelft.nl (Cezar Rodolfo Wedig Reinbrecht), marcelo.brandalero@b-tu.de (Marcelo Brandalero), S.Hamdioui@tudelft.nl (Said Hamdioui), M.Taouil@tudelft.nl (Mottaqiallah Taouil)

Although hardware implementations are much more efficient, many systems still perform RSA in software to avoid the area overhead. [14]. Boneh *et al.* [15], Bao *et al.* [16], and Lenstra [17] have all proposed methods that could break RSA implementations using any kind of fault attack technique. Today, due to the increased complexity and high performance requirements of integrated circuits, current state-of-the-art protections might not be suitable anymore [18, 19]. Therefore, new approaches are required to protect software implementations of RSA.

We can group the protections for software RSA implementation in three: i) prevention, ii) detection, and iii) redundancy. In prevention, the protective countermeasures aim to prevent the injection of a fault in the integrated circuit. Passive shields are an example for this category [1]. These shields cover the circuit with extra metal layers in order to make it hard for electromagnetism or light to inject a fault. Note that such a protection is limited, as they cannot prevent other threats like voltage or clock-based fault attacks. The detection countermeasures on the other hand aim to monitor the system behavior to identify when a fault attack happens. Ngo *et al.* [20] proposed an active shield to detect laser and EM-based attacks. Their active shield holds the encoded data and the integrity of it can be checked to determine if it is faulted or not. However, similar to the passive shields, this countermeasure does not protect against voltage and clock-based attacks. Another detection approach is based on the monitoring of physical disturbances using sensors. Examples include detecting light, voltage fluctuations or clock frequency variations [1]. The main issue with such countermeasures is their coverage, as a single sensor typically can only detect one type of fault attack. An overall secure system would require many sensors to protect against the different types of attack, which is not very practical due to area and power costs. The third type of detection approaches is based on redundancy. Redundancy can be added in time [21, 22] or in space [23, 24]. A basic time-based redundancy repeats an operation later in time and compares both results [25]. A more complex version is to calculate the inverse operation and compare the result with the input [26]. Other versions that validate the operation with a completely different operation also exists [27, 28]. Giraud *et al.* [29] presented one way of achieving this by adding two additional redundant modular multiplications to RSA and verifying the integrity using several checksums. In space-based approaches, either multiple operations can run in parallel at the same time using additional hardware [23, 24] or different checking mechanisms are added, such as error correction codes/parity checks [24]. However, in such implementations, a final checker or voter is required to decide if a fault was injected. In general, redundancy-based techniques can protect a system against all kinds of fault injection attacks, but come with two major drawbacks. First, the entire security is compromised if the checker is glitched. Second, these countermeasures have a high overhead: typically, twice the execution time or twice the hardware usage. Therefore, better low-cost countermeasures are needed that do not only protect against all types of fault attacks, but are also robust against faults attacks targeted at them.

In our previous work [30], we proposed a novel and efficient strategy to detect all kinds of fault attacks. The countermeasure evaluates the order of instructions of an RSA decryption and detects a fault if the expected instruction flow is broken. We implemented this detector using a recurrent neural network (RNN), learning simply from the observations of instruction sequences without faults. In this study, we extend our previous work by introducing a detailed analysis of the impact of different parameters regarding instruction sequences (e.g., sequence length) on security. In addition, this study presents two other implementations that meet the proposed concept. They are content addressable memory (CAM) and Bloom filter (BF). Thereafter, we make a comprehensive comparison among the implementations, showing the trade-offs related to security, hardware costs, and application flexibility. Lastly, we performed robustness analysis

to compare how secure is each implementation when a fault targets the detector itself. Therefore, our contributions can be summarized as follows:

- Analytical evaluation of instruction flow sequences in terms of security and cost.
- Proposal of a content addressable memory based fault attack detector.
- Proposal of a Bloom filter based fault attack detector.
- A location-based fault analysis to assess the vulnerability of a processor.
- A comprehensive trade-off analysis between RNN and the two new detectors, in terms of detection rate of faults and hardware costs.

We organize the rest of the paper as follows. Section 2 provides information about the tools used by the detectors elaborated in this article. Section 3 describes the threat model considered in this work. Section 4 presents the detectors. Section 5 presents the experiments and results. Finally, Section 6 concludes and discusses the significance of the obtained results.

2. Background

This section provides the background required to understand the working principles of the detectors in the context of this study. First, Subsection 2.1 presents the recurrent neural networks. Thereafter, Subsection 2.2 describes the Content Addressable Memories. Finally, Subsection 2.3 describes the Bloom Filters.

Before describing the detectors, we will briefly define the context here (more details can be found in Section 4). In our study, we consider instruction sequences (i.e., collection of a couple of instructions) as inputs to the detectors. The output on the other hand is a binary value that states whether the instruction sequence contains a fault or not.

2.1. Recurrent neural network (RNN)

A recurrent neural network is a neural network that is capable to understand the relation of a dataset over time. For example, given a sequence of elements, an RNN can be trained to predict the next element in such sequence. This functionality is achieved by incorporating previous state information using a feedback loop [33].

An RNN consists of one or more layers each containing at least one RNN cell. Figure 1 (a) shows how such a cell processes information over time. In our case, the information is the about to be executed instructions that constitute an instruction sequence. During the first time step, the RNN cell takes the first instruction from the sequence as input at time t_0, computes the dot product of it with the weight matrix $\boldsymbol{W}$. The result is used as an input to a nonlinear function f (usually tanh). In the following time steps (i.e., t_1, t_2, etc.) the same operation is repeated for the next instructions of the sequence. The only difference here is that the output of the previous time step is concatenated with the next instruction input before the dot product computation. The RNN consists of several layers, each containing multiple cells.

The neural network contains two phases: i) a *design* phase (consisted of training and validation); and an ii) *evaluation phase*. During training, the network is trained using a reference dataset (e.g., instruction sequences without any faults). During validation, the training performance is evaluated with another reference dataset. The expectation is that the RNN is able to

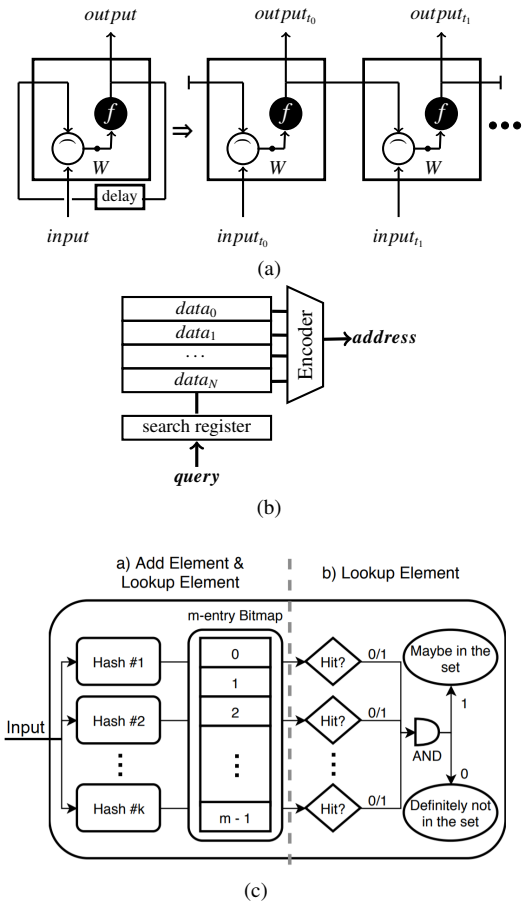


Figure 1: Background architectures: (a) RNN cell and its unrolling in time [31]; (b) CAM memory [32]. (c) Bloom Filter.

predict the next instruction, given the past few instructions in the sequence. Finally, the RNN is employed in the field, where it uses (instruction sequences that may include faulty instructions) as input in the *evaluation phase*.

2.2. Content addressable memory (CAM)

Content addressable memory is a special type of memory, where you query for the location of a specific content [32]. In other words, it receives data as input and outputs the respective address in the memory in case such content exists. CAMs are typically used in network applications due to the dynamic information flowing in networks. For example, if multiple destinations use the same path, a CAM is able to store all these destinations on the same address. As a result, for this type of application, the memory usage and performance are optimized.

A typical CAM is illustrated in Figure 1 (b). When a query consisting of an instruction sequence is supplied, all rows are searched for a matching instruction sequence. If there is a match, the matching row address is encoded and supplied as an output. There are two characteristics of

such an architecture: (i) it finds out whether the query is stored (and its address if that is the case) and (ii) it accomplishes this typically in a single clock cycle.

The same phases used in RNN is also applicable to the CAM (i.e., *design phase* and the *evaluation phase*). During the *design phase*, the reference dataset (i.e., instruction sequences without any faults) is stored in the CAM. In the *evaluation phase*, instruction sequences with faulty instructions could be queried. If the input is found in the CAM, the address of the matched query is retrieved. Due to its deterministic behavior, there is no need for a validation for CAM.

2.3. Bloom filter (BF)

A Bloom filter is a probabilistic data structure that can be used quickly to check whether an element belongs to a predefined set or not. A BF can be implemented either in software or in hardware, and it contains the following key components: i) k different hash functions, and ii) an m-entry bitmap (representing a set). The hash functions must be *independent*, *uniformly distributed*, and in order to provide fast operations, they also have a computational cost.

Fig. 1 (c) depicts an example of how an element can be added (step a) and looked up (steps a and b) in a BF. We refer to the operation of adding elements to the Bloom Filter as *design phase*, while the look-up operation is referred to as *evaluation phase*. At the beginning of the *design phase*, all entries in the bitmap are first set to 0. Next, each item of the training dataset, i.e., instruction sequences without any faults, is processed by the k different hash functions (see step a). Each hash function produces an integer in the range $[0, m-1]$, which is used as an index in the m-entry bitmap. During the *design phase*, the k bitmap positions indexed by the hashes are set to 1. This phase ends when all instances of the training dataset are computed, and the bitmap memory is filled.

Similarly, during the *evaluation phase*, hash values of instruction sequences that may include faulty instructions are computed using the same k hash functions (see step a). However, in contrast to the *design phase*, the resulting indices are now used to read the content of the bitmap memory. Hence, the k positions are accessed and their values are fed into an *AND* operation, as shown in step b of Figure 1 (c). If the *AND* returns a 1, there is a probability (depending on k and m) that the input element belongs to the valid set. Note that for BF, false positives are possible. If the *AND* returns a 0 instead, the element is definitely not in the valid set.

Notice that a BF never produces false negatives. In other words, it never identifies an element as a non-member of the set when it actually does. In the context of this work, this property ensures that a non-faulty instruction sequence will never be detected as faulty. Additionally, the accuracy obtained in the *evaluation phase* can be pre-adjusted using the parameters k and m. Many works provide mathematical estimates for accuracy bounds based on m and k, and we base our analysis on the results of [34]. Provided that the hash functions are perfectly random, the false positive rate (FPR) (i.e., probability that a malicious behavior is mistakenly identified as non-malicious) can be estimated by Equation 1 [34]:

$$\text{FPR} = (1 - e^{-\frac{kn}{m}})^k, \tag{1}$$

where n represents the number of instruction sequences that are part of the set. This equation allows the parameters to be configured for different levels of accuracy and costs, and hence, enables a fast and cheap implementation.

3. Threat & Attack Model

This section describes the threat model. It discusses the target algorithm, target implementation, fault models, fault exploitation methods, and fault attacks.

3.1. Target Algorithm - RSA

The target algorithm we aim to protect in this work is RSA [35]. It is an asymmetric cryptographic algorithm that consists of three phases (see Figure 2): key generation, encryption and decryption. In the key generation phase, a public and a private key are generated based on two large primes p and q (Steps 1-6 of Figure 2). The public key k_{pub} consists of e (the public exponent) and n (product of two large prime numbers p and q), while the private key k_{pr} consists of d (the private exponent) and n. The public key is available to everyone and can be used to send encrypted messages to the receiver. In RSA, the encryption is performed by exponentiating the message m with the public exponent e, which results in a ciphertext c (Step 9 of Figure 2). When the ciphertext is received, the receiver can decrypt the original message m_{dec} by exponentiating the ciphertext with the private exponent d, which is only available to the receiver (Step 11 of Figure 2).

Sender		**Receiver**	
	I. Key Generation		
		1 :	Generate two large primes (p, q)
		2 :	$n \leftarrow p \times q$
		3 :	$\Phi(n) \leftarrow (p-1) \times (q-1)$
		4 :	Generate $e \in \{1, 2, \ldots, \Phi(n)\}$ $\mid gcd(e, \Phi(n)) = 1$
		5 :	$d \leftarrow e^{-1} \bmod \Phi(n)$
		6 :	$\{k_{pub}, k_{pr}\} \leftarrow \{(e, n), (d, n)\}$
	$\xleftarrow{k_{pub} = (e,n)}$	7 :	
	II. Encryption		
8 :	Generate message m		
9 :	Encrypt m: $c \leftarrow m^e \bmod n$		
	$\xrightarrow{10 : \quad c}$		
	III. Decryption		
		11 :	Decrypt c: $m_{dec} \leftarrow c^d \bmod n = m$

Figure 2: The RSA cryptosystem [36].

Algorithm 1 square-and-multiply (for RSA decryption) [36].

Input: Private key $k_{pr} = (d, n)$ and ciphertext c
Output: Decrypted message $m_{dec} = c^d mod\ n$

1: Let $d_b = \{d_{b_0}, d_{b_1}, \ldots, d_{b_B}\}$ be the base-2 (bit) representation of d
2: $m_{dec} \leftarrow c$
3: **for** $i \leftarrow B - 1$ downto 0 **do**
4: $\quad m_{dec} \leftarrow m_{dec}^2 \mod n$ ▷ square in every step
5: $\quad$ **if** $d_{b_i} = 1$ **then** ▷ branch condition
6: $\quad\quad m_{dec} \leftarrow (m_{dec} \times c) \mod n$ ▷ multiply if the key bit is 1
7: $\quad$ **end if**
8: **end for**

Algorithm 2 Chinese remainder theorem (for RSA decryption) [36].

Input: Private key $k_{pr} = (d, n)$, two (secret) large primes (p, q) and ciphertext c
Output: Decrypted ciphertext $m_{dec} = m$

1: $m_p \leftarrow c^d \mod p$ ▷ smaller modulo exponentiation for p
2: $m_q \leftarrow c^d \mod q$ ▷ smaller modulo exponentiation for q
3: $a_p \leftarrow q^{-1} \mod p$ ▷ auxiliary calculation for p
4: $a_q \leftarrow p^{-1} \mod q$ ▷ auxiliary calculation for q
5: $m_{dec} \leftarrow ([q \times a_p]m_p + [p \times a_q]m_q) \mod n$ ▷ combination

The security of RSA depends on the selection of prime numbers p and q. As n is public, as shown in Step 2, an attacker may obtain p and q by brute-forcing the factorization of n. To overcome this, large numbers are used, typically in the order of 1024 bits and beyond. As a consequence, the selection of large numbers affects the encryption and decryption performance (Steps 9 and 11). To speed them up, different algorithms have been proposed. In this paper, we look at two algorithms for the decryption. The first algorithm uses square-and-multiply (SAM) (see Algorithm 1) for the exponentiation. SAM decomposes the exponentiation in a series of iterative square operations and potential multiplications based on the binary representation of the key. As a result, this algorithm has logarithmic time complexity. The second algorithm is based on the Chinese remainder theorem (CRT) (see Algorithm 2). This method first computes the exponentiation for two smaller numbers p and q as modulo (typically also using SAM). Thereafter, it linearly combines these results to obtain the actual exponentiation in the larger modulo n. The performance gain in CRT comes from this task division. This algorithm typically uses the *extended Euclidean algorithm* [37] to calculate modular inverses of p and q. CRT provides a performance advantage when big integers are used.

3.2. Target Implementation - RISC-V ISA

In this work, we consider an RSA implementation in software compiled for the RISC-V instruction set architecture (ISA), in particular, the 32-bit base architecture (RV32I). RISC-V is an open source ISA that contains four core instruction formats, either 32, 64, or 128 bits and several optional extensions [38]. The four core instruction formats are **R-type:** used for arithmetic and logical operations where three registers are involved; **I-type:** used for short immediate and loads; **S-type:** used for loads, stores, and branches; and **U-type:** used for long immediate and unconditional jumps. There are several format extensions, such as floating point (extension F)

or compressed instructions (extension C), which aim to provide flexibility to adapt the processor according to the needs of the target application.

The base 32-bit instruction set RV32I includes 47 instructions, which can be grouped into six types if we consider two additional variants with respect to the four core instruction formats. These two extra formats are the **B-type** (used for conditional branches, which is a variation of the S-type) and **J-type** (used for unconditional jumps, which is a variation of the U-type). Figure 3 illustrates the format of the different instruction types. In all of them, the least significant seven bits are used as opcode. Aside from the U-type and J-type formats, bits 12 to 14 are referred to as function 3 (f3) field. These two fields determine the functionality of the instruction. In the R-format, which is used for operations where three registers are involved, an additional function 7 (f7) field is used to specify extra functionality details. This field is seven bits wide, from bit 25 to 31. Six of these bits are always 0. The value of the 30^{th} bit is used to further clarify the instruction. For example, an f3 value of $\{000\}_2$ may indicate addition or subtraction. If the 30^{th} bit equals 0 (i.e., f7 equals $\{0000000\}_2$), the operation equals an add, otherwise (when f7 is $\{0100000\}_2$) a subtraction is performed. Note that we used the opcode, function f3, and 30^{th} bit located in function 7 as inputs to the neural network, as these define an instruction. Without loss of generality, we do not consider out-of-order and speculative executions in this work.

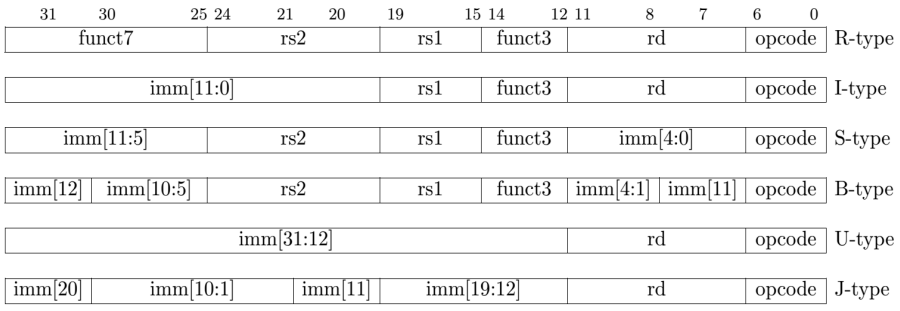


Figure 3: RISC-V RV32I instruction formats [38].

3.3. Fault Models

We used two sets of fault models. Both sets assume that an attacker can inject faults in the different parts of the systems such as the main memory, processor in general and specific parts of the processor such as instruction buffer during the sensitive operation. In addition, we assume that attackers have full observability, i.e., they can observe inputs (ciphertext), as well as faulty and fault-free outputs (decrypted message).

The first set of fault models can be used by an attacker to determine vulnerable locations in the system. Therefore, they focus on faults in different locations of the system. There are four location-based fault models:

1. *One fault in memory (OM)*. This fault model represents one bit flip in the main memory.
2. *One fault in processor (OP)*. In this fault model, one random bit flip occurs in any part of the processor.

3. *Multi-bit fault in memory (MM).* In this fault model, multiple random bit flips occur in any part of the main memory (from one to four, where the latter is set to limit the simulation times). Note that these faults may fall in any place, and hence, they are not necessarily concentrated in the same or adjacent memory row.

4. *Multi-bit fault in processor (MP).* In this fault model, multiple random bit flips occur in any part of the processor (from one to four, again for the same reason).

The second set was introduced by Koylu *et al.* in [30], in order to evaluate the instruction fault detection capabilities of a detector. Here, the fault models represent different types of faults that alter instructions that can take place for example in the instruction memory or the instruction buffer of the processor. It contains five types of fault models:

1. *Single bit fault model.* This fault model represents a single bit flip that may happen in any bit of the instruction.

2. *Single byte fault model.* A byte fault refers to multiple bit flips within a single byte of the instruction. Any fault that provokes a change in a random byte falls into this fault model.

3. *Branch-to-opposite fault model.* This fault model contains bit flips that change a branch instruction to the opposite branch instruction. As we consider the RISC-V ISA, these bit flips must happen in the f3 field. As such, the instructions are swapped between *branch equal<->branch not equal*, *branch less than<->branch greater or equal* and *branch less than unsigned<->branch greater or equal unsigned*.

4. *Instruction-to-instruction fault model I.* This fault model extends the previous, by also including the faults resulting in the change of other instructions to each other. This change can be in the same format (e.g., from branch equal to branch greater) or into different formats (e.g., from branch to add). One constraint in this fault model is that only a branch instruction can be glitched into another branch instruction. The reason for this is that when a non-branch instruction is glitched into a branch, it is very easy to detect the fault as the control flow of the program breaks and the program typically crashes.

5. *Instruction-to-instruction fault model II.* This fault model is the same as the variation I, but without the branch constraint.

Note that the instruction changing fault models can also cover popular software attacks such as instruction skips, code injection, buffer overflow, and code reuse [39]. However, we do not formally investigate them in this paper.

3.4. Fault Exploitation Methods

The exploitation methods show how vulnerabilities can be exploited to break a cryptosystem by injection of faults. The objective is to identify three main aspects: "how many faults are needed?", "where should the faults occur?", and "what type of faults are needed?". Two popular methods against RSA are considered in this paper. These are referred to as Bellcore and Bao.

3.4.1. Bellcore Threat Model

One of the first fault exploitation methods against RSA is "Bellcore" [15]. This theoretical study demonstrated that some particular faults allow malicious parties to break Chinese remainder theorem-based RSA implementations (i.e., obtain the key). The attack aims at inserting a fault into one of the smaller modulo exponentiation (see Algorithm 2) to provoke an erroneous result. By comparing the wrong output with the correct output from fault-free decryption, the key can be mathematically retrieved. To understand the attack in more detail, let's revisit the smaller modulo exponentiation $m_p \equiv c^d \mod p$ and $m_q \equiv c^d \mod q$ (see also Algorithm 2). There are two coefficients (a, b) that satisfy the following three properties:

p1. $m_{dec} = m \equiv a \times m_p + b \times m_q \mod n$

p2. $a \equiv 1 \mod p, a \equiv 0 \mod q$

p3. $b \equiv 0 \mod p, b \equiv 1 \mod q$

Let's assume that a fault occurred during decryption which affected m_p only (see line 2 at Algorithm 2). As $m_p \equiv c^d \mod p$, property *p1* will change and the faulty m'_p can be expressed by Equation 2. In case the fault-free m_{dec} is available, a differential calculation can be made; this is shown in Equation 3. From this equation the value of q could potentially be derived, as shown in Equation 4. In case the result of Equation 3 is not divisible by p, the value of q can be retrieved. Hence, RSA can be easily broken as $n = p \times q$. Later, Lenstra et. al showed that the correct message m_{dec} is even not needed to break the cryptosystem [17].

$$m'_{dec} \equiv a \times m'_p + b \times m_q \mod n \tag{2}$$

$$m_{dec} - m'_{dec} = (a \times m_p + b \times m_q) - (a \times m'_p + b \times m_q) = a(m_p - m'_p) \tag{3}$$

$$\gcd\{a(m_p - m'_p), n\} = \begin{cases} q, \text{ if } m_{dec} - m'_{dec} \mod p \neq 0. \\ n, \text{ otherwise.} \end{cases} \tag{4}$$

3.4.2. Bao Threat Model

A second popular fault exploitation method against RSA is presented in Bao *et al.* [16]. In this study, two threats are introduced. Both are based on the idea of introducing *bit faults* to leak one bit of the secret exponent d at a time. To understand why this strategy works, the decryption operation can be rewritten in such a way that the key bits are used independently from each other. This is shown in Equation 5. In this equation, d_i presents the i^{th} bit of the key and N the bit length of the modulus n. The values t_i depend on the ciphertext c as shown in Equation 6.

$$m_{dec} \equiv t_{N-1}^{d_{N-1}} \times t_{N-2}^{d_{N-2}} \times \ldots \times t_1^{d_1} \times t_0^{d_0} \mod n \tag{5}$$

$$t_i \equiv c^{2^i} \mod n : i \in \{0, 1, \ldots, N-1\} \tag{6}$$

The first attack injects a bit fault into the ciphertext. More specifically, one of the t_i's are made faulty by one bit. It can be quickly observed from the equation that only one of the N terms of Equation 5 will differ from the correct decryption. Hence, when you divide the faulty output with the fault-free output, either a 1 is expected (when the involved key bit is 0) or the ratio between

them (when the involved key bit is 1). This is shown in Equation 7. Note that the first condition on this equation, when $d_i = 0$, means that $m'_{dec} \mod n \equiv m_{dec} \mod n$. Thus, no information can be gained in this case. For the other case however, an attacker can calculate all possible 1 bit faults on t_i's, and compare it with the result of $\frac{m'_{dec}}{m_{dec}}$, which are both assumed to be accessible. When a match is found, the attacker infers both i and that $d_i = 1$. This attack is then repeated to find other bits of the secret exponent d.

$$d_i = \begin{cases} 0, \text{ if } \frac{m'_{dec}}{m_{dec}} \equiv 1 \mod n \\ 1, \text{ if } \frac{m'_{dec}}{m_{dec}} \equiv \frac{t'_i}{t_i} \mod n \end{cases} \tag{7}$$

In the second attack, the bit fault is injected into the secret exponent d. Namely, d_i is made faulty. In a similar way, Equation 8 is now applicable. In the equation, both cases $d_i = 0$ and $d_i = 1$ leak information. The secret bit is 0 or 1 when the division of the correct and faulty decrypted messages results in t_i or $\frac{1}{t_i}$, respectively. This attack is then repeated to find the other bits.

$$d_i = \begin{cases} 0, \text{ if } \frac{m'_{dec}}{m_{dec}} \equiv t_i \mod n \\ 1, \text{ if } \frac{m'_{dec}}{m_{dec}} \equiv \frac{1}{t_i} \mod n \end{cases} \tag{8}$$

3.5. Fault Attacks

Fault injection attacks on RSA try to successfully retrieve the key by applying exploitation methods like the ones presented in the previous subsection. Several techniques can be used to inject faults into a circuit, such as voltage underfeeding or clock glitching. For example, Barenghi *et al.* [3] used voltage underfeeding to retrieve the key of a software RSA implementation using both Bellcore and Bao methods. To perform Bellcore's method, they corrupted the load instructions. To perform Bao's method, they changed *branch* instructions to the opposites, during square-and-multiply (Algorithm 1). To elaborate further on this, the check condition that determines whether a multiplication should be performed (which depends on the key bit) is typically implemented with a *branch not equal* instruction. When this condition is changed to a *branch equal*, the effect is equal to having a bit fault in the secret exponent d. Hence, the vulnerability conditions described by the Bao threat model can be exploited. Schmidt *et al.* [10] successfully applied the Bellcore threat model-based attack using an EM spark, which was able to disturb only one of the modulo computations.

It is important to note here that the conditions of the aforementioned threats can be realized in a number of ways. Although not explicitly aimed at RSA implementations, many studies show different fault injection techniques that can potentially be used to achieve Bellcore and Bao exploitation methods. For example, Amiel *et al.* [2] tampered with the clock signal of a smartcard to break data encryption standard (DES). The study by Kim *et al.* [40], introduced Rowhammering to flip bits in memory cells by continuously reading data from DRAMs. The importance is that the attacker does not even have to obtain physical access to the attacked device. Skorobogatov *et al.* [41] illustrated the same effect by using a flash lamp, whereas Agoyan *et al.* [11] used a laser. Finally, Schmidt *et al.* [42] used ultraviolet light to attack advanced encryption standard (AES).

The location of where the faults take place is important. In the context of a processor running an RSA decryption, the attacks can target multiple places to succeed in realizing Bellcore and Bao threats. These include; (i) injecting faults to the main memory to change associated stored

values [40], (ii) injecting faults to the instructions to change the operations and/or the interpretation of the data [3], and (iii) injecting faults into the internal signals of the processor to change the interpretation of intermediate results [43]. Our protection focuses on faults that affect the instructions directly or indirectly in the instruction buffer, which is explained next.

4. Instruction Flow-based Detectors

This section presents a detailed analysis on instruction flow-based detectors. Subsection 4.1 explains the concept of how faults can be detected by observing the program flow at instruction level. Thereafter, Subsection 4.2 provides a mathematical relation between the detector's detection capability (i.e., security) versus its cost. Subsequently, Subsection 4.3 presents our general methodology to design an instruction-flow based detector. Finally, Subsection 4.4 presents different types of detectors: RNN, CAM, and BF.

4.1. Concept

Our aim is to design a detector that works in parallel to the processor, with the aim of detecting faulty instructions. This requires two main elements: a way to extract meaningful information from the instructions, and an algorithm to detect faults in them. This subsection focuses on the ways to extract meaningful information.

Every program runs a specific sequence of instructions that is dictated by its algorithm. Depending on the data, a program can have multiple execution flows, which creates multiple valid/fault-free/correct instruction sequences. Note that we use these terms interchangeably. If a fault occurs, it is very likely that a valid sequence becomes corrupted. This can lead to erroneous computations or even crashes. Therefore, faults can be detected by investigating the validity of the sequence of executed instructions. The more instructions an instruction sequence contains, the easier it is to detect the fault in general, as the order of the instructions is more unique. In contrast, if the instruction sequence consists of a single instruction only, the probability that a faulty instruction is still valid is much larger: such as the case when an add instruction is faulted to a subtract instruction.

A number of studies have investigated a similar concept; we refer to them as *control flow integrity checking* [44, 45, 46, 47, 19, 48]. These studies divided the instructions of a program into blocks and protected them using signature-based integrity checks at the end of each block. Even though this approach would theoretically determine faulty instructions, it has major drawbacks: i) a fault injected into the signature checker at the end of a block will render the countermeasure inefficient, ii) there are typically no security dependencies between the blocks and hence if one check is bypassed, checks in the subsequent blocks cannot detect that, iii) the exhaustive listing of all possible program behavior is costly, and iv) they typically require modifications to the processor [39]. Our previous work addressed these shortcomings by simply evaluating sequences of non-faulty instructions (i.e., the last couple of fetched instructions) continuously in the processor [30] to determine irregularities when there are faulty ones.

This concept is shown in Figure 4, where w_l denotes the window length (i.e., the size of the instruction sequence, which contains the last w_l instructions that are being checked), while s_l represents the sliding length, i.e., how many instructions are skipped between the sequences. Note that an s_l of 1 represents an overlap of w_l-1 instructions between two consecutive sequences (see Figure 4). To prevent instructions from not being checked, s_l must be equal to or smaller than w_l. In the following subsection, we provide detailed analysis regarding w_l and s_l.

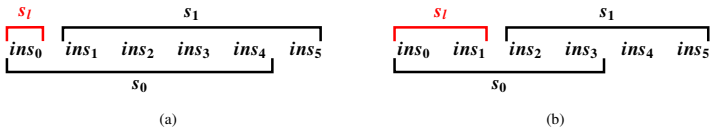


Figure 4: Illustration of two cases of instruction sequences, with the following parameters as examples: (a) $w_l = 5$, $s_l = 1$ and (b) $w_l = 4$, $s_l = 2$. Two sequences (s_0 and s_1) are indicated for both cases.

4.2. Analysis

The window w_l and sliding length s_l impact the security and cost as follows:

- If the probability of randomly changing an instruction to another valid instruction by a fault injection attack is p, changing an instruction such that it still matches a valid sequence of more than one instruction is q, where $q \ll p$. In this case, an adversary's success rate is reduced to q.
- Furthermore, when instruction sequences are validated (e.g., by a detector) instead of single instructions, the success rate of an attack (i.e., changing a complete sequence to another one) becomes $Q = q^{w_l}$. Hence, we have $Q \ll q$, which also means that the bigger w_l is, the lower the probability of an attack to succeed.
- An instruction can be validated multiple times, as $1 \leq s_l \leq w_l$ holds. Specifically, the instructions are validated in approximately $l = \left\lceil \frac{w_l}{s_l} \right\rceil$ different sequences. The lower s_l, the more overlap of instructions in different sequences, and hence, more redundant checks are performed. This further reduces the adversary success probability Q' as $Q' < Q$.

Based on these observations, a countermeasure can be designed to protect the system by evaluating instructions sequences. A large w_l and small s_l is expected to increase the security, but also come with a higher implementation cost. In order to analyze the trade-off between security versus cost, we propose two evaluation metrics to represent them. The security can be expressed in how often an instruction gets checked and how difficult it is to change an instruction without getting detected by the detector. The latter implies that a bit causing a fault in a valid sequence must lead to another valid sequence in order to go undetected. Hence, we use the average Hamming distance between the different sets as a security metric as shown in Equation 9.

$$Security = \frac{l \times 2}{N \times (N-1)} \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} \frac{HD(seq_i, seq_j)}{B}. \tag{9}$$

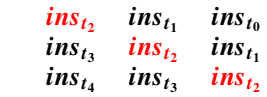
In this equation, l denotes how often the same instruction is checked in different sequences, HD the hamming distance between two sequences seq_i and seq_j normalized with respect to the number of bits in a sequence B, and N the number of different instruction sequences which can be approximately represented by $N \approx (I - w_l)/s_l$. Here, I represents the total number of instructions. Lastly, to calculate the average HD between the instruction sequences the equation is normalized by the number of different sequence pairs (i.e., $[N \times (N-1)]/2$).

The security in Equation 9 is directly proportional to the number of instruction checks, and thus a larger w_l and smaller s_l increases the security. However, at the same time using such values will increase the required storage and computational capacity. The storage can be expressed by the number of bits that need to be stored, while the computation complexity by the number of

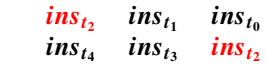
instructions processed in parallel at a given time. We integrate both concepts in a single cost metric, as shown in Equation 10.

$$Cost = N \times B \times (w_l \times l), \tag{10}$$

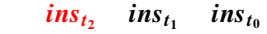
In this equation, the storage requirement equals the product of number of sequences N and the number of bits in each sequence B. For the computational capacity, we consider the number of instructions that are processed from the moment a new instruction is part of the instruction sequence under process until the moment it is not part of an instruction sequence. Figure 5 provides an example for the instruction ins_{t_2}. The amount of instructions that are processed while ins_{t_2} is being checked equals $w_l \times l$, which can also be represented as $i_p = w_l \times \left\lceil \frac{w_l}{s_l} \right\rceil$. For part (a) of the figure with $w_l = 3$ and $s_l = 1$ this equals $i_p = 9$. For part (b) with $w_l = 3$ and $s_l = 2$ equals $i_p = 6$, and for part (c) with $w_l = 3$ and $s_l = 3$ equals $i_p = 3$.



(a) $w_l = 3$, $s_l = 1$ ($i_p = 9$ instructions)



(b) $w_l = 3$, $s_l = 2$ ($i_p = 6$ instructions)



(c) $w_l = 3$, $s_l = 3$ ($i_p = 3$ instructions)

Figure 5: Number of processed instructions before the last instruction in the first sequence (indicated by red) is released.

We calculated the *security* and *cost* metrics for the RSA decryption with and without CRT. Both algorithms were coded in C language and compiled for the RISC-V ISA. The metrics were evaluated for $\{w_l, s_l\} \in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$, with $s_l \leq w_l$. Next, we only present the results for CRT case in Figure 6 and Figure 7, since the results for the non-CRT are very similar.

Figure 6 shows the impact of w_l and s_l on the security, while Figure 7 shows the impact on the cost metrics. The larger w_l the higher the *security*. However, the *cost* increases faster than the security. This analysis ultimately shows that increasing the instruction sequence length brings more protection but at a higher price. In other words, w_l must be chosen carefully, being high enough to provide adequate protection, while low enough to avoid a high cost. Increasing parameter s_l, denoted as slide in the figure, on the other hand leads to a faster reduction of cost as compared to the security.

In our previous work, we selected $w_l = 5$ and $s_l = 1$. This seems a reasonable selection as the cost is not too high. We made this selection for the RNN based on training and validation performance (with trial-and-error), as is customary while working with neural networks. This selection of a low w_l value also fitted our goal of lowering the cost of a hardware implementation [30]. In this work, we use the same w_l and s_l values for CAM and BF implementations.

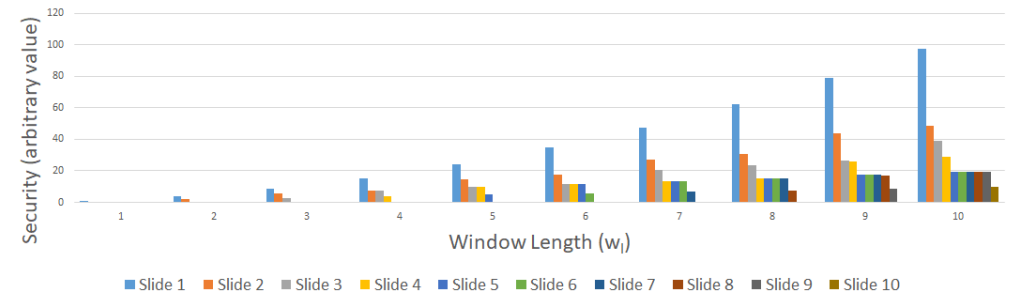


Figure 6: Protection level analysis for CRT implementation.

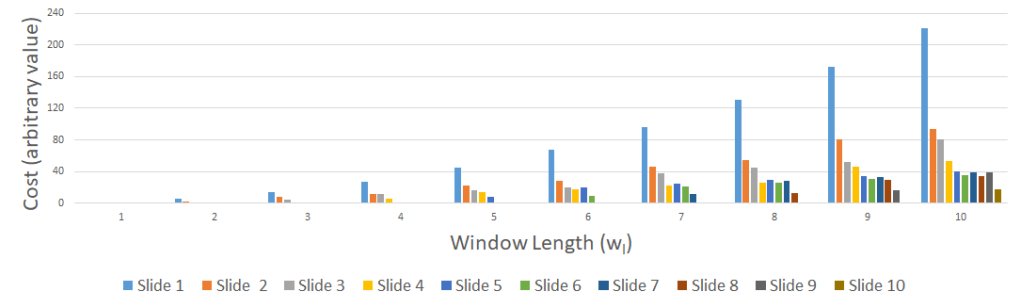


Figure 7: Cost analysis for CRT implementation.

4.3. Design

The background section (Section 2) presented different tools that can be used to build instruction-flow-based detectors. All these methods use a similar design approach consisting of two phases: *design* and *evaluation*. Both phases are described next.

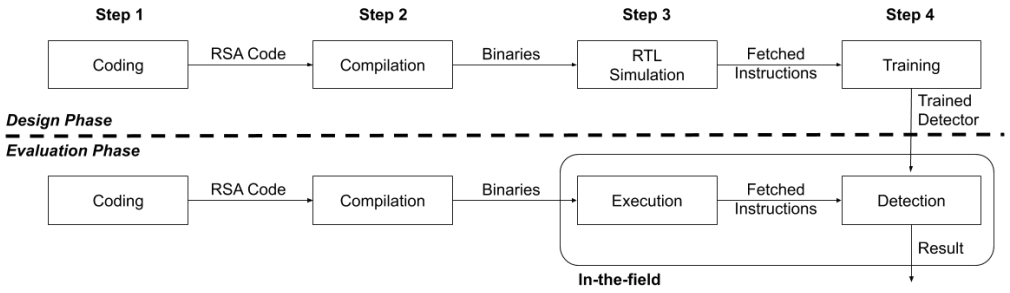


Figure 8: Design methodology of instruction-flow based detectors

4.3.1. Design Phase

The design phase consists of four steps, which are illustrated at the top of Figure 8. The aim of *step 1* (Coding) is to create a software code or a program, in this case, the RSA decryption. For

this, we implemented two RSA decryptions in C, one with and one without CRT. Both implementations generate random public/private keys and a ciphertext. The software implementation also contains the extended Euclidean algorithm (EEA) needed for CRT. Moreover, both implementations use square-and-multiply (SAM) for exponentiation. Next, *step 2* (Compiling) compiles the created programs to the target implementation. In this work, we employed the riscv_gcc (version 7.1) [49] compiler to generate the binaries. These binaries contain all assembly instructions required to generate the instruction sequences.

In *step 3* (Simulation), we load the generated binaries into the instruction memory of the RISC-V processor. This processor is written in register-transfer level (RTL) and is part of a system-on-chip (SoC) containing the processor, cache memory and peripherals. Next, we simulate this SoC using QuestaSIM from Mentor Systems [50] and Incisive from Cadence Design Systems [51]. During simulation, instructions are fetched from the instruction memory and are executed. The simulator saves the sequence of executed instructions into a file as output and marks those related to the decryption. Lastly, *step 4* (Training) uses this file of instruction sequences to build a training dataset and perform the training process. When the training is complete, the RNN is able to predict the next instruction from previous ones. To determine whether a next instruction is valid with an RNN, we need an additional parameter called $conf_{thr}$, which we set as the lowest expectance probability of all instructions in the *validation set* [30]. Consequently, if a run-time instruction has a lower expectance probability than $conf_{thr}$, the detector considers it faulty. The CAM saves different correct sequences into its table and BF fills the bitmap positions after the hash calculations of correct sequences (see also Figure 1.(c)). Consequently, both are able to determine whether an instruction sequence is valid or not. At the end of this *design phase*, the detector is ready to be tested for use in the field, which constitutes the *evaluation phase*.

4.3.2. Evaluation Phase

The evaluation phase also consists of four steps, as illustrated in the bottom part of Figure 8. The first two steps are similar to the *design phase*, where the software programs (Coding) are compiled to the target processor (Compiling). In *step 3* (Execution), the SoC is tested in the field. During this step, the processor fetches the instructions from its memory. In parallel, those fetched instructions are copied to a buffer to be used by the detector. Note that at this moment, the system can be exposed to a fault attack. Lastly, *step 4* (Detection) represents the detector evaluating the sequence of instructions and providing fault detection results. In the presence of an alarm (i.e., when the detector identifies an invalid sequence), the system can take some action. This is beyond the scope of this work; however, some examples are restarting the operation or the system and changing secret keys.

To test the effectiveness of the detector, we simulated the processor with the aforementioned fault models (see Subsection 3.3) that were applied in the testbench. We argue though that this is not different from testing in a real environment. The reason is that our detectors are solely trained on instruction sequences of fault-free operations. Hence, the detectors are up-front not aware of any faults. Therefore, the detection results are bias-free and give an idea about the performance against unknown or future attacks.

4.4. Implementation

In this section, we describe the hardware implementation of the three different detectors: RNN, CAM and BF. All three detectors are intended to be used as a hardware module in the

SoC, with inputs (i.e., fetched instructions) provided from the instruction buffer. We omit the description of the implementation of the RNN as it is provided in [30]. A detailed description of the other two methods is provided next.

4.4.1. CAM

The hardware implementation of the CAM-based detector, which is illustrated in Figure 9, consists of three major components: a buffer, table, and an finite state machine (FSM) controller. The function of the buffer is to collect the last five fetched instructions in a first in-first out (FIFO), which outputs a $5 \times 32 = 160$ bit signal. After each newly fetched instruction, the content of the FIFO is updated by shifting in the newly fetched instruction.

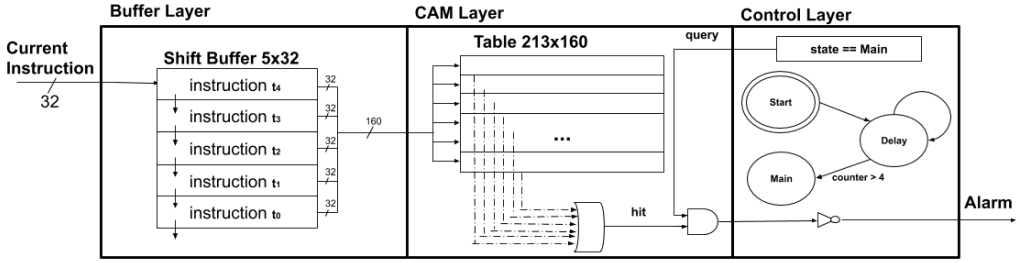


Figure 9: Hardware implementation of CAM-based detector

A CAM is used to identify if the instruction sequence consisting of five instructions is a valid sequence or not. The internal logic of the CAM compares this input with every existing entry. The output is 1 (hit) if there is such an entry and hence a valid sequence, or 0 (miss) otherwise when the sequence is invalid.

The FSM controller makes sure that the initial sequence of five instructions is properly initialized and synchronizes the communication between the buffer and CAM to ensure that a fault check happens each time a new instruction is fetched. When a fault is detected, an alarm signal is raised. Such a signal could for example be an interruption request (IRQ) to the CPU).

4.4.2. BF

As mentioned in Equation 1, the fault detection rate of a BF depends on three parameters: the number of hash functions k, expected number of elements n that equals the number of different valid instruction sequences, and number of entries in the bloom filter memory m.

From the simulations, we identified that $n = 213$ for the CRT implementation, and $n = 63$ for the non-CRT implementation. The analysis for these given n values and varying k and m is illustrated in Figure 10. The plots immediately show that a higher m reduces the false positive rate (FPR). To have a low FPR, we used $m = 512$ bits in our experiments. In terms of k, $k = 2$ results in the smallest FPR for the CRT implementation. For the non-CRT case, $k = 3$ has the lowest FPR. However, to have a single design for both cases we select $k = 2$ as this gives overall the lowest FPR for both designs. The hashes that we select are *fnv* [52] and *murmur* [53]. Each of these hash functions takes a 32-bit input and produces a hash value in a single clock cycle, thus also enabling one-cycle lookups. With all these parameters selected, we used the architecture shown in Figure 1.(c) to make the hardware implementation.

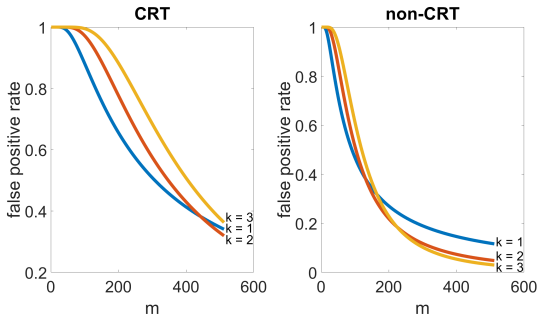


Figure 10: False positive rate analysis for the Bloom filter

5. Experimental Results

In this section, we describe the experimental setup, the performed experiments, and obtained results. In the last part, we also evaluate the hardware overhead of the proposed detectors in terms of area and power.

5.1. Setup

We implemented the RSA implementations using 12-bit keys (without loss of generality) to speedup simulations. Table 1 shows the design parameters of RNN, CAM and Bloom filter. We used 750 fault-free decryptions to train the RNN, whereas the *validation set* consists of 250 fault-free decryptions. We obtained $conf_{thr}$ values of 3.65 for the Chinese remainder theorem (CRT) and 12.69 for the non-CRT case after the training, using the *validation set*. The CAM contains 213 entries (i.e., 213 different instruction sequences as multiple instances of the same instruction sequence have been added once only) for the CRT and 63 for the non-CRT case (for reference: the binary of the decryption implementation contains 174 instructions for CRT and 44 for non-CRT). For the Bloom filter, we have k=2 hashes, n=213 sequences for CRT, n=63 sequences for non-CRT, and m=512. Also note that in contrast to RNN which monitors only 11 bits of the instructions, CAM and BF monitor all 32 bits.

We evaluate the overhead of the detectors by synthesizing and mapping them on an FPGA using as target the device 10AS066N3F40ELG from the ARRIA 10 family [55]. The processor and the detector are implemented in hardware and the clock frequency was set to 25MHz.

5.2. Performed Experiments

In this subsection we are going to describe the experiments that are used to accomplish the following goals: (i) make a vulnerability analysis on various fault attack locations, and (ii) evaluate the detector's performance of attacks on the most vulnerable location. In total there are three experiments. In the first experiment, we assess how vulnerable the processor is. In the second experiment, we evaluate and inject faults only in part of the processor (i.e., instruction buffer) to increase the attack's success rate. This allows us to compare the detection accuracy of the three detectors better in the third experiment. Each experiment is further described next:

Table 1: Design parameters

RNN	
parameter	**value**
s	5
s_l	1
#used instruction bits	11
validation ratio	25%
optimizer	adam [54]
loss function	categorical crossentropy
metrics	accuracy
batch size	100
epochs	100
dropout	RNN layer: 0.1 (normal, recurrent)
CAM	
parameter	**value**
s	5
s_l	1
#used instruction bits	32
Bloom filter	
s	5
s_l	1
#used instruction bits	32
n	213 (CRT) - 63 non-CRT
m	512
hash functions	fmv, murmur

5.2.1. Experiment 1 - Vulnerability Assessment of Processor

The aim of the first experiment is to analyze vulnerable parts of a processor against faults. For this, we injected faults into random locations (including the memory and the processor parts), using the first set of fault models (see Subsection 3.3). These fault models are one fault in memory (OM), one fault in processor (OP), multi-bit fault in memory (MM), and multi-bit fault in processor (MP). The binary of the complete program has a size of 10.4 kB, from which 696 bytes contain instructions related to the decryption for CRT which equals $696/4 = 174$ instructions. Similarly, the non-CRT decryption part has a size of 176 bytes). Since the total memory size is 64 kB, only 1.06% of the memory contains the target program (0.26% for the non-CRT).

For each fault model, a *test set* is used that contains single correct decryption and 10000 runs with injected faults. In some trials, the simulator was not able to inject a fault. This happens for example when a fault is injected into an undefined signal. These cases have not been considered in the results.

Note that this experiment covers all possible cases that can lead to an incorrect decryption result. These include: (i) glitching the memory where the program instructions and data are stored, (ii) glitching the instructions in the instruction buffer of the processor, and (iii) glitching the internal processor signals to corrupt intermediate results (like the arithmetic logic unit (ALU) input or output). Hence, the result of this experiment allows an efficiency comparison of different fault injection strategies.

5.2.2. Experiment 2 - Vulnerability Assessment of Instruction Buffer

Injection faults randomly in the processor and memory typically lead to a low success attack rate. To increase this, and hence to to be able to compare the performances of the detectors better, we repeat the previous experiment but limit the location of faults to the instruction buffer

only (see attack case (ii) in Subsection 5.2.1) and use the *single bit* fault model only (see Subsection 3.3). We have created 2000 different decryptions and injected bit flips into one or more instructions.

5.2.3. Experiment 3 - Detector Evaluation

In this experiment, we evaluate our detectors (RNN, CAM, and BF) by injecting faults to the instruction buffer only (see attack case (ii) in Subsection 5.2.1). We use all the fault models in the second set (see Subsection 3.3): *single bit*, *single byte*, *branch-to-opposite*, *instruction-to-instruction-I/II* fault models.

5.3. Results

Next, we present the results of the three experiments.

5.3.1. Experiment 1

The results of the first experiment are shown in Table 2. The results are here presented based on the observed outputs for the different fault models. Four different output categories are observed. They are: i) expected; ii) crash; iii) successful; and iv) exploitable. Expected behavior means that the fault did not have an impact on the output of the decryption (omitted in the table). A crash represents corruption in the execution, which makes the processor hold. Successful means that the fault(s) have changed the decryption output. Note that not all changes in the output of the decryption can be exploited by an attack. Exploitable indicates the cases where faults caused exploitable outputs by Bellcore and/or Bao's methods (see Subsection 3.4).

Table 2: Vulnerability Assessment of Processor (Experiment 1).

fault	*CRT*			*non-CRT*		
model	**crash**	**successful**	**exploitable**	**crash**	**successful**	**exploitable**
OM	0.07%	0.16%	0.12%	0.00%	0.04%	0.00%
OP	0.94%	2.11%	1.50%	0.16%	0.18%	0.06%
MM	0.26%	0.73%	0.49%	0.02%	0.02%	0.02%
MP	3.40%	4.27%	2.75%	1.65%	1.41%	0.01%

The results shows than only a small percentage of the 10000 fault injection trials for each model (i.e., OM, OP, MM, MP) leads to exploitable cases. This shows that randomly injecting faults without considering the precise location is not very effective.

Another observation is that attacking the processor in general yields better results than attacking the memory. This is primarily because the actively used memory is small in contrast to the attack surface. Thus, the majority of the faults do not create an effect. Therefore, from the results, the best approach is to target the processor with multiple faults.

Note that there are some 0.00% entries in Table 2. These results are due to a low possibility of occurrence. As an example, for the OM non-CRT case, there are some successful instances. Such an instance can lead to a vulnerability, but it did not in our sample. The same applies to the case of no crashes.

5.3.2. Experiment 2

Evaluating the detector based on the first experiment would require many runs for a fair comparison, as the only limited cases lead to exploitable cases. Therefore, we focus in this experiment on injecting fault in the instruction buffer only. The results of this experiment are presented in Table 3, and are represented in a similar manner as the results of Experiment 1.

Table 3: Vulnerability Assessment of Instruction Buffer (Experiment 2).

CRT			*non-CRT*		
crash	**successful**	**exploitable**	**crash**	**successful**	**exploitable**
34.29%	47.76%	28.84%	33.63%	49.97%	10.02%

The results show that an incomparably larger percentage of the faults create vulnerabilities when the instruction buffer is targeted. This shows that attacking the instruction buffer is a much more effective and time-efficient fault injection strategy. Another observation is that the number of exploitable instances is smaller in percentage in non-CRT, compared to the CRT case. One contributing factor is that the non-CRT case cannot be exploited with Bellcore.

This experiment indeed shows that glitching the instruction buffer is a better strategy to compare the performance of the three detectors. However, it must be noted that a more localized fault attack generally requires more knowledge of the design and better fault-attack equipment.

5.3.3. Experiment 3

The results of the third experiment are provided in Tables 4, 5, 6 for RNN, CAM, and BF-based detectors respectively. The results are grouped in three classes: fault, decryption, and security detection. The fault detection column contains the rate of traces that were detected by the detector. The decryption detection column includes the ratio of test cases we can protect, i.e., fault detection rate plus the cases where the faults did not affect the decryption result. The security detection column contains the ratio of traces that could not be attacked, i.e., the decryption detection rate plus the cases where Bellcore and Bao exploitation methods (see Subsection 3.4) did not work. The table also shows additional information by also looking at the number of faults that have been injected. For example, for fault model 1 we have injected a single bit fault in one instruction (the first line where $f = 1$) and a single bit fault in two or more instructions (the second line where $f > 1$).

Table 4: Detector Evaluation of RNN-based detectors ((Experiment 3 part (a))

		fault		*decryption*		*security*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f = 1$	0.35	0.28	0.70	0.54	0.75	0.88
	$f > 1$	0.65	0.62	0.71	0.69	0.82	0.95
2	$f = 1$	0.60	0.55	0.80	0.69	0.83	0.95
	$f > 1$	0.88	0.84	0.91	0.86	0.93	0.99
3	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
4-I	$f = 1$	0.91	0.90	0.95	0.91	0.97	0.99
	$f > 1$	0.99	0.99	0.99	0.99	1.00	1.00
4-II	$f = 1$	0.88	0.90	0.95	0.91	0.96	0.99
	$f > 1$	0.99	0.99	0.99	0.99	0.99	1.00

The results show that CAM has a 100% detection accuracy for all cases, BF almost 100% in all cases and RNN only has a high detection rate when fault models are applied that change instructions. Note that RNN provides some detection even for bit and byte fault models. This is because (i) some faults hit on instruction locations that are learned by the RNN and (ii) some data faults can still disrupt the instruction flow, such as a change in the jump location in a branch instruction. Overall, the deterministic methods result in higher accuracy.

Table 5: Detector Evaluation of CAM-based detectors ((Experiment 3 part (b))

		fault		*decryption*		*security*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
2	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
3	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
4-I	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
4-II	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00

Table 6: Detector Evaluation of BF-based detectors ((Experiment 3 part (c))

		fault		*decryption*		*security*	
fault model	**#faults (f)**	**CRT**	**non-CRT**	**CRT**	**non-CRT**	**CRT**	**non-CRT**
1	$f = 1$	0.99	0.99	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
2	$f = 1$	0.99	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
3	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
4-I	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00
4-II	$f = 1$	1.00	1.00	1.00	1.00	1.00	1.00
	$f > 1$	1.00	1.00	1.00	1.00	1.00	1.00

We also evaluated our detector against 10000 correct decryptions that are not part of *training*, *validation set* and *test set* to realize the impact of false positives. In none of the cases false positives have been detected and hence, the false positive is 0% for all three detectors.

5.4. Hardware overhead

All detectors have been synthesized on an FPGA technology and compared to the Ariane core. Table 7 shows the area overhead in percentage for the three detector implementations: RNN (including a single RNN cell), CAM (for both CRT and non-CRT cases) and BF. Available resources in absolute value are indicated in parenthesis. Note that the RAM comparison only considers internal components of the Ariane core, such as caches and buffers (implemented as SRAM blocks). Hence, no external memory is considered.

Table 7: Area overhead of three detector implementations: RNN, CAM, and BF

tool	**slice LUTs** (4182)	**slice registers** (273)	**block RAM tiles** (32)
RNN	15.57%	2.17%	1300.67%
RNN (1 cell)	1.92%	0.17%	162.58%
CAM (CRT)	2.05%	0.07%	61.35%
CAM (non-CRT)	0.55%	0.03%	26.97%
BF	0.51%	0.17%	154.61%

As observed in the table, the RNN-based detector is the most expensive implementation. In addition to requiring a lot of memory, this implementation leads to increased overhead in the processor. However, if desired, this implementation can be employed by a single RNN cell. This would reduce the overhead significantly but increase the computation time significantly.

In contrast, both CAM implementations have a much lower overhead, especially the non-CRT case, due to limited number of instruction sequences. The memory overhead in CAM depends linearly on the amount of different instruction sequences that have to be protected. BF implementation on the other hand is a middle ground between RNN and CAM with respect to LUT and register usage. However, BF has the same overhead for both CRT and non-CRT implementations.

6. Conclusion and Discussion

In this study, we extended our instruction sequence-based fault detector for software RSA with two new implementations. We tested their effectiveness and efficiency using realistic fault models. The results show in general that the detectors were able to detect faults that affect an instruction or instruction sequence. We conclude the paper with the following observations:

- **Functionality:** Our experimental results show that detectors obtained a 100% accuracy prediction for correct decryptions. Note that this also works for decryptions with different key lengths as the main part of the decryption contains a key-dependent loop. Increasing or decreasing the loop size will not change the order of the instruction flow (except on the boundary of the loop iterations). Similarly, the extended Euclidean algorithm, which computes the modular multiplicative inverse of a number that is used in CRT also consists of a limited number of instructions within a loop. Hence, the detectors are able to learn this very well.

- **Security:** Experimental results show that our detectors attain a nearly 100% detection rate for faults that change the instructions for all implementations. For CAM and BF implementations, nearly any fault in the instruction buffer could be detected. Note that for successful exploitation, the attacker needs in addition to obtaining an exploitable faulty output also the correct output. Obtaining the correct output is possible, but is difficult (e.g., the attacker needs to have access to the platform and run the same decryption without fault injection). Getting this correct output is not considered in this discussion, as well as a strong attacker that is able to find and continuously exploit one of the very few uncovered cases.

  Another important security feature of our detector is the checking mechanism. As the detector checks for fault in every fetched instruction, one successful glitch on this check is not enough to break the system for two reasons. First, the instruction that is glitched and the evaluation that checks its integrity have to be glitched both at the right moments. Second, when the flow is disrupted, it is likely that the detection will catch faults in consecutive instructions as a single instruction is checked multiple times in different sequences and as a change in instruction flow will be detected as well. This can be observed from near-perfect detection rates.

- **Weaknesses:** One vulnerability of the RNN-based detector is the confidence threshold $conf_{thr}$ value. If an attacker manages to glitch and lower the $conf_{thr}$ value, more faulty

decryptions would be seen as correct by the detector. A designer may therefore choose to harden this by considering multiple copies of $conf_{thr}$, or use some other form of redundancy like parity checks. The detectors use only input from the instruction buffers to identify faults. Hence, faults injected into the memory that affect data or faults injected inside the processor (e.g., an add instruction could be executed as a subtraction) might not be detectable. However, the results of Experiment 1 in Table 2 clearly show that the probability of successful injection faults that are exploitable is marginal. Note that even when a fault injection is successful, only a single bit of the key is typically leaked for Bao. Hence, it would be very time consuming to recover the complete key by applying such an approach.

- **Robustness:** Besides glitching the RSA instructions, an attacker could also glitch the detector itself. To analyze the resiliency of our detector implementations, we conducted a number of experiments. Each experiment consisted of 20 trials, in which we evaluated the detector performance under a random fault configuration, using 1000 correct decryptions (the ones that are not part of any set, see Subsection 5.3) and the 2000 faulty decryptions of *Instruction-to-instruction fault model II* (see Subsection 3.3), both with CRT as a case study. For the RNN, we injected a random bit or byte fault to the network weights. For the CAM, we again injected a random bit or byte fault to one of the entries, simulating an attack against the memory. For the BF, we injected a bit fault to one of the BF entries to simulate a memory glitch. Moreover, as we wanted to simulate faulty hash calculations, we injected a bit fault to the same place of each input. Each of the trials yielded a similar result: a large number of false alarms for correct decryptions, but also a considerable increase in fault detection capability. Most importantly, the results show that the attacker cannot gain an advantage by trying to glitch the detector, except for disrupting the operation for correct decryptions. This is a very unique property of our detector, compared to the state of the art.

- **Comparison:** As the experimental results in Section 5.3 indicate, deterministic methods (CAM and BF) provide more coverage, and create less overhead. On the other hand, RNN provides a flexibility that is not directly possible in CAM or BF. By setting the value of $conf_{thr}$, a user can directly determine the security level of the system. There is a possibility to adjust the detection rate in relation to Equation 1, by changing values k and m (as n is fixed). However, this false positive rate is not exact, and does not give the granularity of setting the $conf_{thr}$.

  Moreover, the scalability of the CAM solution cannot be guaranteed. Theoretically, a branch-extensive application can produce a great number of different instruction sequences. This favors the BF solution over CAM, as in essence, it proposes a way to compress the stored data.

- **Uniqueness:** The detectors presented in this study can be compared with *control flow integrity checkers* [39]. However, as mentioned in Section 4.1, we use a much simpler instruction validation structure than creating control flow blocks based on program jumps. To elaborate further, we can make a comparison with the study presented in [19], which can be considered as a baseline control flow integrity method. In that study, the authors rely on both using encrypted instructions and comparing block signatures with pre-computed versions. Although exhaustive pre-computing theoretically covers all valid and invalid program flows (while our observation-based method is not exhaustive), such an approach

creates storage and computational overhead, as well as attacks to the architecture itself are still a viable strategy: a fault replacing the final signature/message authentication code (MAC) check can cause faulty instructions to be executed. If that is the case, it is not possible to retroactively detect a faulty block further in line. By replacing pre-calculated control flow blocks with valid instruction sequence observations, we can detect faults later, even when we miss the original fault occurrence. Finally, our detectors do not require any modifications to the processor, or any encryption/storage of encrypted data, as we only need to create an interface with the instruction buffer.

In order to address the limitations of the control flow integrity checker proposed in [19], a number of variations have been proposed. First, the authors in [56] aimed to address the single point of failure (MAC check) issue. As such, the authors proposed to append execution history to the current instruction, making the decrypted instruction faulty (thus detectable) if there was a fault previously. However, especially for complex programs, this further complicates the control flow, as there is a need to adjust for different branches during execution. Second, the authors in [48] aimed to remove the need to modify the processor. As a result, the authors put their integrity checker as a module interacting with the processor and the memory, similar to our solution. However, their proposal does not address other shortcomings of control flow studies, as it does one check per instruction block. Lastly, the authors in [18] aimed to address the complexity of pre-computing by eliminating the need for determining all possible branch locations beforehand. They used masks to connect sequences of instructions to the previous ones and encrypt them together. To accomplish that however, they require an extension to the ISA, damaging general applicability.

- **Generality and flexibility:** Although we demonstrated the detection results for two different implementations of RSA, our detector can be used for a variety of applications. This includes RSA algorithms with protections against other attacks such as side-channel analysis, other software crypto algorithms as AES, triple-DES and Ellyptic curve cryptography (ECC). Not only crypto algorithms, but also other security sensitive applications such as banking and secure boot can be protected. The detector can actually be used for multiple applications when the weights of RNN, table of CAM, or bitmap of BF are adjusted at runtime.

- **Applicability:** We have developed our detectors to work in conjunction with the processor. When employed, the hardware of the detector will be static. When the user wants to run a specific security sensitive application however, the operating system can load the associated weights to RNN, or memory entries to CAM and BF. Another possibility is to include these in the binary. If on the other hand, the device supports reconfigurable hardware, different RNNs (e.g., with a different number of layers, cells, etc.) or CAM and BF with different memory sizes can be employed with each application.

  One point of concern is the operation of detectors during processor interruptions. When the processor receives an interruption signal, the execution context changes. Such a signal can be used to halt the operation of our detector. As such, the detector will not process fetched interruption handling instructions. When the interruption ends and the previous context is restored, the detector can continue its work. Our detector should only work when the processor executes security sensitive applications and switched off by the operating system otherwise.

7. Acknowledgements

The authors would like to thank Ahmet Çağrı Bağbaba for his support with the Incisive simulator.

References

[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, The sorcerer's apprentice guide to fault attacks, Proceedings of the IEEE 94 (2) (2006) 370–382.
[2] F. Amiel, C. Clavier, M. Tunstall, Fault analysis of dpa-resistant algorithms, in: International Workshop on Fault Diagnosis and Tolerance in Cryptography, Springer, 2006, pp. 223–236.
[3] A. Barenghi, G. Bertoni, E. Parrinello, G. Pelosi, Low voltage fault attacks on the RSA cryptosystem, in: 2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), IEEE, 2009, pp. 23–31.
[4] B. Giller, Implementing practical electrical glitching attacks, Black Hat Europe (2015).
[5] A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pellicioli, G. Pelosi, Low voltage fault attacks to AES, in: 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), IEEE, 2010, pp. 7–12.
[6] N. Selmane, S. Guilley, J.-L. Danger, Practical setup time violation attacks on AES, in: 2008 Seventh European Dependable Computing Conference, IEEE, 2008, pp. 91–96.
[7] A. Djellid-Ouar, G. Cathebras, F. Bancel, Supply voltage glitches effects on cmos circuits, in: International Conference on Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006., IEEE, 2006, pp. 257–261.
[8] L. Zussa, J.-M. Dutertre, J. Clédiere, B. Robisson, A. Tria, et al., Investigation of timing constraints violation as a fault injection means, in: 27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France, Citeseer, 2012, p. 11.
[9] S. Govindavajhala, A. W. Appel, Using memory errors to attack a virtual machine, in: IEEE Symposium on Security and Privacy, Vol. 5, 2003.
[10] J.-M. Schmidt, M. Hutter, Optical and EM fault-attacks on CRT-based RSA: Concrete results, na, 2007.
[11] M. Agoyan, J.-M. Dutertre, A.-P. Mirbaha, D. Naccache, A.-L. Ribotta, A. Tria, How to flip a bit?, in: 2010 IEEE 16th International On-Line Testing Symposium, IEEE, 2010, pp. 235–239.
[12] S. Burnett, S. Paine, The RSA security's official guide to cryptography, McGraw-Hill, Inc., 2001.
[13] P. Kaliyamoorthy, A. C. Ramalingam, Qmlfd based RSA cryptosystem for enhancing data security in public cloud storage system, Wireless Personal Communications 122 (1) (2022) 755–782.
[14] A. S. Alkalbani, T. Mantoro, A. O. M. Tap, Comparison between RSA hardware and software implementation for wsns security schemes, in: Proceeding of the 3rd International Conference on Information and Communication Technology for the Moslem World (ICT4M) 2010, IEEE, 2010, pp. E84–E89.
[15] D. Boneh, R. A. DeMillo, R. J. Lipton, On the importance of checking cryptographic protocols for faults, in: International conference on the theory and applications of cryptographic techniques, Springer, 1997, pp. 37–51.
[16] F. Bao, R. H. Deng, Y. Han, A. Jeng, A. D. Narasimhalu, T. Ngair, Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults, in: International Workshop on Security Protocols, Springer, 1997, pp. 115–124.
[17] A. K. Lenstra, Memo on RSA signature generation in the presence of faults, Tech. rep. (1996).
[18] O. Savry, M. El-Majihi, T. Hiscock, Confidaent: Control flow protection with instruction and data authenticated encryption, in: 2020 23rd Euromicro Conference on Digital System Design (DSD), IEEE, 2020, pp. 246–253.
[19] R. De Clercq, J. Götzfried, D. Übler, P. Maene, I. Verbauwhede, Sofia: software and control flow integrity architecture, Computers & Security 68 (2017) 16–35.
[20] X. T. Ngo, J.-L. Danger, S. Guilley, T. Graba, Y. Mathieu, Z. Najm, S. Bhasin, Cryptographically secure shield for security ips protection, IEEE Transactions on Computers 66 (2) (2016) 354–360.
[21] L. Anghel, M. Nicolaidis, Cost reduction and evaluation of a temporary faults-detecting technique, in: Design, Automation, and Test in Europe, Springer, 2008, pp. 423–438.
[22] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, F. Regazzoni, Countermeasures against fault attacks on software implemented AES: effectiveness and cost, in: Proceedings of the 5th Workshop on Embedded Systems Security, ACM, 2010, p. 7.
[23] R. Karri, K. Wu, P. Mishra, Y. Kim, Fault-based side-channel cryptanalysis tolerant rijndael symmetric block cipher architecture, in: Proceedings 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, IEEE, 2001, pp. 427–435.
[24] R. Karri, G. Kuznetsov, M. Goessel, Parity-based concurrent error detection of substitution-permutation network block ciphers, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2003, pp. 113–124.

[25] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, J.-P. Seifert, Fault attacks on RSA with CRT: Concrete results and practical countermeasures, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2002, pp. 260–275.
[26] A. Boscher, H. Handschuh, E. Trichina, Fault resistant RSA signatures: Chinese remaindering in both directions., IACR Cryptology ePrint Archive 2010 (2010) 38.
[27] A. Shamir, Method and apparatus for protecting public key schemes from timing and fault attacks, uS Patent 5,991,415 (Nov. 23 1999).
[28] D. Vigilant, RSA with CRT: A new cost-effective solution to thwart fault attacks, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2008, pp. 130–145.
[29] C. Giraud, An RSA implementation resistant to fault attacks and to simple power analysis, IEEE Transactions on computers 55 (9) (2006) 1116–1120.
[30] T. C. Koylu, C. R. W. Reinbrecht, S. Hamdioui, M. Taouil, RNN-based detection of fault attacks on RSA, in: 2020 IEEE International Symposium on Circuits and Systems (ISCAS), 2020, pp. 1–5.
[31] C. Olah, Understanding lstm networks (Aug 2015).
URL `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`
[32] K. Pagiamtzis, A. Sheikholeslami, Content-addressable memory (cam) circuits and architectures: A tutorial and survey, IEEE journal of solid-state circuits 41 (3) (2006) 712–727.
[33] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, S. Khudanpur, Recurrent neural network based language model, in: Eleventh annual conference of the international speech communication association, 2010.
[34] A. Broder, M. Mitzenmacher, Network applications of bloom filters: A survey, Internet Mathematics 1 (4) (2004) 485–509. doi:10.1080/15427951.2004.10129096.
[35] R. L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Communications of the ACM 21 (2) (1978) 120–126.
[36] C. Paar, J. Pelzl, Understanding cryptography: a textbook for students and practitioners, Springer Science & Business Media, 2009.
[37] E. W. Weisstein, Euclidean algorithm (2002).
[38] A. Waterman, K. Asanovic, The risc-v instruction set manual-volume i: User-level isa-document version 2.2, RISC-V Foundation (May 2017) (2017).
[39] R. de Clercq, I. Verbauwhede, A survey of hardware-based control flow integrity (cfi), arXiv preprint arXiv:1706.07257 (2017).
[40] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, O. Mutlu, Flipping bits in memory without accessing them: An experimental study of dram disturbance errors, in: ACM SIGARCH Computer Architecture News, Vol. 42, IEEE Press, 2014, pp. 361–372.
[41] S. P. Skorobogatov, R. J. Anderson, Optical fault induction attacks, in: International workshop on cryptographic hardware and embedded systems, Springer, 2002, pp. 2–12.
[42] J.-M. Schmidt, M. Hutter, T. Plos, Optical fault attacks on AES: A threat in violet, in: 2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), IEEE, 2009, pp. 13–22.
[43] I. Verbauwhede, D. Karaklajic, J.-M. Schmidt, The fault attack jungle-a classification model to guide you, in: 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, IEEE, 2011, pp. 3–8.
[44] R. Vemu, J. A. Abraham, Ceda: Control-flow error detection through assertions, in: 12th IEEE International On-Line Testing Symposium (IOLTS'06), IEEE, 2006, pp. 6–pp.
[45] J. R. Azambuja, M. Altieri, J. Becker, F. L. Kastensmidt, Heta: Hybrid error-detection technique using assertions, IEEE Transactions on Nuclear Science 60 (4) (2013) 2805–2812.
[46] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, H. Quinn, S-seta: Selective software-only error-detection technique using assertions, IEEE transactions on Nuclear Science 62 (6) (2015) 3088–3095.
[47] G. Di Natale, O. Keren, Nonlinear codes for control flow checking, in: 2020 IEEE European Test Symposium (ETS), IEEE, 2020, pp. 1–6.
[48] J.-L. Danger, A. Facon, S. Guilley, K. Heydemann, U. Kühne, A. S. Merabet, 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), IEEE, 2018, pp. 529–536.
[49] The risc-v embedded gcc (Jul 2017).
URL `https://gnu-mcu-eclipse.github.io/toolchain/riscv/`
[50] Questa® advanced simulator.
URL `https://www.mentor.com/products/fv/questa/`
[51] Incisive enterprise simulator.
URL `https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html`
[52] G. Fowler, L. C. Noll, K.-P. Vo, D. Eastlake, T. Hansen, The fnv non-cryptographic hash algorithm, Ietf-draft

(2011).
[53] A. Appleby, Murmurhash 2.0 (2008).
[54] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).
[55] Intel Arria 10 FPGAs, accessed at 23-10-2019. Available at: https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html.
[56] M. Werner, T. Unterluggauer, D. Schaffenrath, S. Mangard, Sponge-based control-flow protection for iot devices, in: 2018 IEEE European Symposium on Security and Privacy (EuroS&P), IEEE, 2018, pp. 214–226.