Evaluating the Code Encryption Effects on Memory Fault Resilience

Evaluating the Code Encryption Effects on Memory Fault Resilience

Riccardo Cantoro, Nikolaos I. Deligiannis, Matteo Sonza Reorda
Politecnico di Torino, Dip. Automatica e Informatica
Torino, Italy
{riccardo.cantoro|nikolaos.deligiannis|matteo.sonzareorda}@polito.it

Marcello Traiola, Emanuele Valea
LIRMM (Université de Montpellier - CNRS)
Montpellier, France
{traiola|valea}@lirmm.fr

***Abstract*—In most safety-critical systems, the robustness and the confidentiality of the application code are crucial. Such code is generally stored into Non-Volatile Memories (NVMs) that are prone to faults (e.g., due to radiation effects). Unfortunately, faults affecting the instruction code result very often into Silent Data Corruption (SDC). This condition lets faults remain undetected and it can lead to undesiderable errors that may compromise the system functionality. Thus, it is desirable that the system is able to detect faults affecting the code memory. To overcome this issue, designers often resort to expensive error detection/correction mechanisms. Furthermore, they also adopt memory encryption techniques to prevent unauthorized, hence malicious, access to the code or to protect it from any unauthorized copy. In this paper, we show that the presence of memory encryption alone is able to strongly reduce the probability of SDC, without the need of implementing expensive error detection.**

We have performed some experiments on the OpenRISC1200 microprocessor in order to evaluate the impact on reliability stemming from different encryption methods.

I. Introduction

Designers of Embedded Systems are used to face many challenges: among them, it is increasingly common to find both safety and security. In fact, in many domains where embedded systems are used (e.g., automotive, biomedical, aerospace) the effects of possible failures due to faults affecting the hardware must be considered, and suitable safety-oriented techniques must be adopted in order to minimize the chances of faults (e.g., acting on the adopted semiconductor technology) or to minimize the impact of their effects (e.g., by introducing redundancy in the design). On the other side, in the last years there has been an exponentially growing interest (and concern) for the security-oriented techniques suited to prevent possible attacks that can deliberately affect the system behavior to change it (in the extreme case, blocking the system, or forcing it to perform illegal operations), or to extract private and/or precious information.

Focusing on security, a common issue which is often faced when designing embedded systems is the one of protecting the application code from possible attacks aimed at stealing it (e.g., to better understand the adopted algorithms for copying purposes, or to allow more intrusive attacks in the following). For this purpose, in these cases the application code is often encrypted when stored in the memory (often a Non Volatile Memory, or NVM) and then decrypted before execution, or when the code is transferred to a RAM [1]. In this way the stored version of the code can hardly be used and understood, even if downloaded from the NVM. A suitable hardware block to execute the decryption is thus included within the circuit and activated when required. Several issues related to efficiently manage an encrypted NVM storing the application code are discussed and solved in [2].

On the other side, since NVMs are prone to errors (e.g., caused by radiation effects), designers adopt redundancy solutions able to detect the occurrence of single- and multiple-bit errors and to possibly correct some of them. The most common solution is based on Error Correction Codes (ECCs) able to detect and possibly correct errors up to a given multiplicity [3]. The adoption of ECCs involves both hardware and performance overhead, because they require computing some extra bits each time the memory is written, and re-computing/checking them each time the memory is read. Moreover, the error detection/correction capabilities of ECCs depend on the number of errors they can manage, and the overhead quickly becomes unacceptable when multiple faults, other than double ones, are considered. Alternative solutions to ECCs have also been explored, based on suitably selecting the instruction opcodes, so that the chances that the processor can trigger an exception when a faulty instruction is fetched are maximized [4]. However, this solution requires acting on the design of the CPU core, which is often impossible for system companies using for their applications processors developed by other companies.

Our work started from the observation that when a fault affects the code memory, it is likely that the decryption mechanism produces a code which triggers some fault detection mechanism in the processor (e.g., an illegal instruction exception). Hence, encrypting the code may turn into a mechanism that detects possible errors in the code memory content. The attractive consequence of this observation is that, if confirmed, it allows an existing mechanism to be used for both security and safety, thus reducing costs. In order to quantify this phenomenon, we performed a number of experiments on the OR1200 processor [5] with different encryption mechanisms and different applications. Results show that by suitably se-

978-1-7281-8731-0/20/$31.00 ©2020 IEEE

lecting the encryption mechanism, a significant percentage of single- and multiple-bit faults can be detected. Hence, a high error detection capability can be achieved without resorting to any Error Correction/Detection mechanism for the memory.

As a result, a major contribution of this paper lies in practically demonstrating that security-oriented mechanisms can be used for detecting at no extra cost hardware faults affecting the code memory.

The rest of the paper is organized as follows. Section II briefly describes the underlying theory of symmetric encryption and why it is applied to embedded system memories. Section III introduces the reference scenario of this work and details the proposed contribution. Section IV illustrates the experimental setup and the related results obtained. Finally, Section V draws the conclusions.

II. Background

In this Section, we provide the background on memory encryption which is required to let the reader appreciate the evaluation proposed in the following of the paper. At first, we summarize the theory of symmetric encryption, introducing the two main categories of encryption mechanisms. Secondly, we explain how these principles are applied to the practical case of securing embedded system memories.

A. Symmetric Encryption

In cryptography, encryption is used to grant confidentiality of exchanged data between a sender and a receiver [6]. Let us imagine a communication scheme where a sender wants to send a private message to a receiver. The incumbent security threat is that an attacker could eavesdrop the communication channel and potentially steal the content of the data. In order to protect the message, the sender and the receiver can rely on *encryption*, whose advantage is making the transmitted message unintelligible for the attacker. In particular, *symmetric encryption* schemes are based on the fact that the sender and the receiver share the same *secret key*.

The plaintext message m is encrypted by the sender using the secret key k and the encryption function $\mathbf{E}$, in order to produce the ciphertext $c = \mathbf{E}(k, m)$. The ciphertext c is sent on the communication channel, but no attacker is able to deduce m from c without knowing the secret key. The receiver can easily retrieve the plaintext m using the secret key and the decryption function $\mathbf{D}$. Indeed, $m = \mathbf{D}(k, c)$.

The couple $(\mathbf{E}, \mathbf{D})$ is called *cipher*. A cipher is defined as *semantically secure*, if it is not possible for an attacker to retrieve the plaintext merely observing the ciphertex. Most common ciphers can be classified into two categories: *block ciphers* and *stream ciphers*.

1) Stream Ciphers: stream ciphers are based on a theoretical cipher, called *One Time Pad* (OTP). In the OTP, the secret key must be as long as the message m. The encryption function is defined as $\mathbf{E}(k, m) = m \oplus k$, and the decryption function as $\mathbf{D}(k, c) = c \oplus k$. Normally, the plaintext and the ciphertext are processed as bitstreams. The key k is called $keystream$ and it is combined one bit at a time with the plaintext to produce the ciphertext stream.

If the keystream is perfectly random (i.e., according to the uniform distribution), the OTP has perfect secrecy. This means that the produced ciphertext is indistinguishable from a random sequence (this is due to the properties of the XOR operation). In this case, it is impossible for an attacker intercepting the ciphertext to derive any information neither on the message nor on the key. However, from a practical point of view, the OTP is not implementable because the sender and the receiver should share a secret key having the same size as the message.

Stream ciphers are the implementation of the OTP. In stream ciphers the keystream is produced by a *Pseudo-Random Generator* (PRG). The PRG takes as input a value k, called *seed* of the stream cipher, and outputs the keystream $S(k)$. The encryption and decryption functions are thus defined as $\mathbf{E}(k, m) = m \oplus S(k)$ and $\mathbf{D}(k, c) = c \oplus S(k)$. Therefore, the secret key that is exchanged between the sender and the receiver is the seed of the PRG (which is much shorter than the resulting keystream). As long as the PRG produces a keystream that is unpredictable, the resulting stream cipher is considered to be secure.

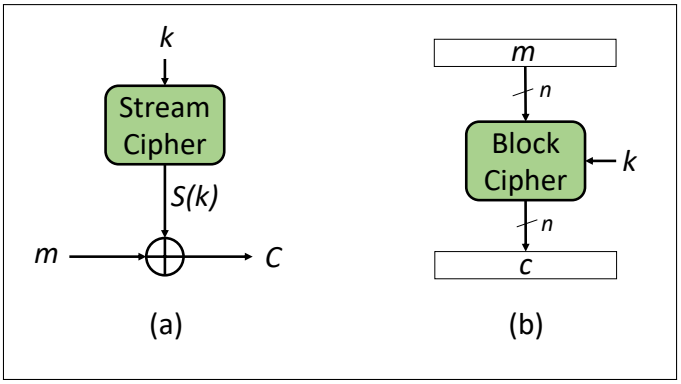

Fig. 1: High-level architecture of: (a) stream cipher; (b) block cipher.

In Fig. 1(a) the high-level structure of a typical stream cipher is represented.

For the purpose of this paper, it is worth pointing out that in stream cipher encryption, we always find a bit-to-bit correspondence between plaintext and ciphertext. In fact, *a one-bit difference on the plaintext leads to the same bit difference on the ciphertext*.

2) Block Ciphers: block ciphers are based on mathematical objects called *Pseudo Random Permutations* (PRPs). They are invertible functions that take as input an n-bit value m and a secret key k, and output an n-bit value c. A PRP is considered secure if it is indistinguishable from a random bijective function on n-bit values. Block ciphers implement a secure PRP. They are made of (i) an encryption function that is able to encrypt a plaintext block into a ciphertext block using a secret key; (ii) a decryption function that performs the inverse operation and retrieves the plaintext block from the ciphertext.

Block cipher functions perform *confusion* and *diffusion* of the plaintext on the ciphertext. This means that two very similar plaintexts result in completely different ciphertexts after the encryption.

In Fig. 1(b) the high-level structure of a typical block cipher is represented.

In block cipher based encryption the plaintext and the ciphertext have a block per block correspondence. This means that *the modification of one bit on the plaintext leads to the perturbation of an entire block of ciphertext*.

B. Memory Encryption

In embedded systems, encryption is widely implemented in order to protect a large variety of assets. All the intellectual properties that are present inside a system could potentially be stolen by an attacker having physical access to the hardware. For this reason, it is important to provide confidentiality of the memory content [7], as well as of the data that is exchanged with the other integrated circuits in the system [8].

NVMs are sensitive elements in all embedded systems. For instance, flash memories can be opened and read out by any attacker having access to the device [9]. Since the application code is typically stored inside external flash memories, the encryption of the memory content is a functionality that is more and more needed by designers. Even when the NVM is part of a single chip also including the CPU, its content could be encrypted to make it more difficult to steal it via some peripheral interface.

The memory encryption model adopted in this paper relies on the symmetric encryption scheme. The sender is the code developer that stores the encrypted code into the NVM. The receiver is the microprocessor that reads out the code from the NVM and loads it into its internal memory (e.g., internal RAM or instruction cache) for being executed. The secret key is known by the code developer and configured inside the microprocessor in a secure way. At system boot, the application code is transferred from the external NVM to the internal RAM and processed on-the-fly by a *decryption module* using the secret key.

III. Reference scenario and proposed evaluation

When a single fault affects the non-encrypted NVM, where the application code is stored, this normally results in the flipping of one instruction bit. If the fault hits the part of the instruction containing the opcode, it may happen that the instruction is transformed into an illegal instruction, causing a failure in the code execution. On the other hand, if the fault affects the part of the instruction containing the operands, there is a higher probability that the resulting instruction is executed by the processor without causing a failure. Thus, the computation carries on with corrupted data, resulting in Silent Data Corruption (SDC).

In this paper, we show that when memory encryption is implemented, faults affecting the application code could produce an effect that may be propagated and amplified by the decryption mechanism. This increases the probability of an illegal instruction condition, thus it decreases the SDC probability. The experiments that we have conducted (discussed in Section IV) show that the SDC reduction heavily depends on the cipher that is used.

For the purpose of this paper we have considered two different ciphers to implement memory encryption:

- *RC4*: a very lightweight stream cipher which is widely used in hardware implementations;
- *Advanced Encryption Standard (AES)*: the most widely used block cipher. It encrypts the plaintext per blocks of 128 bits.

Block ciphers can be implemented according to different *modes of operation*. A mode of operation is an encryption scheme that repeatedly applies a single-block operation in order to securely transform a plaintext that is longer that one encryption block.

In the proposed experiments, we have considered memory encryption with AES configured according to the following modes of operation:

- *Electronic Codebook (ECB)*: blocks are encrypted independently, such that one block of plaintext is mapped to one block of the ciphertext.
- *Cipher Block Chaining (CBC)*: each plaintext block, before being encrypted, is XORed with the ciphertext resulting from the previous block.
- *Cipher Feedback (CFB)*: it behaves like a stream cipher. The ciphertext is obtained as the XOR between the plaintext block and a key block. The key block is obtained by the encryption of the ciphertext resulting from the previous block.
- *Output Feedback (OFB)*: it behaves like a stream cipher. Keystream blocks are created starting from an initialization vector, which are XORed with plaintext blocks in order to generate ciphertext blocks.
- *Counter (CTR)*: it behaves like a stream cipher. Keystream blocks are computed in parallel encrypting blocks produced by a counter. The resulting keystream blocks are XORed with the plaintext blocks in order to compute the ciphertext blocks.

Depending on the error propagation properties of the aforementioned encryption techniques, we can classify them into the following categories:

- **Non-Propagating configurations**: if a fault affects one bit of the encrypted memory, after decryption the fault will not propagate to other bits. Thus, in this case, the resulting code is as much corrupted as it would be without memory encryption. This category includes all the stream ciphers, as well as OFB and CTR modes of operations of block ciphers. For the sake of clarity, in the following of the paper we will refer to these encryption methods with the generic name of *stream ciphers*.
- **Propagating configurations**: if a fault affects one bit of the encrypted memory, during the decryption the fault *propagates to at least one encryption block*. If the processor runs 32-bit instructions and the encryption

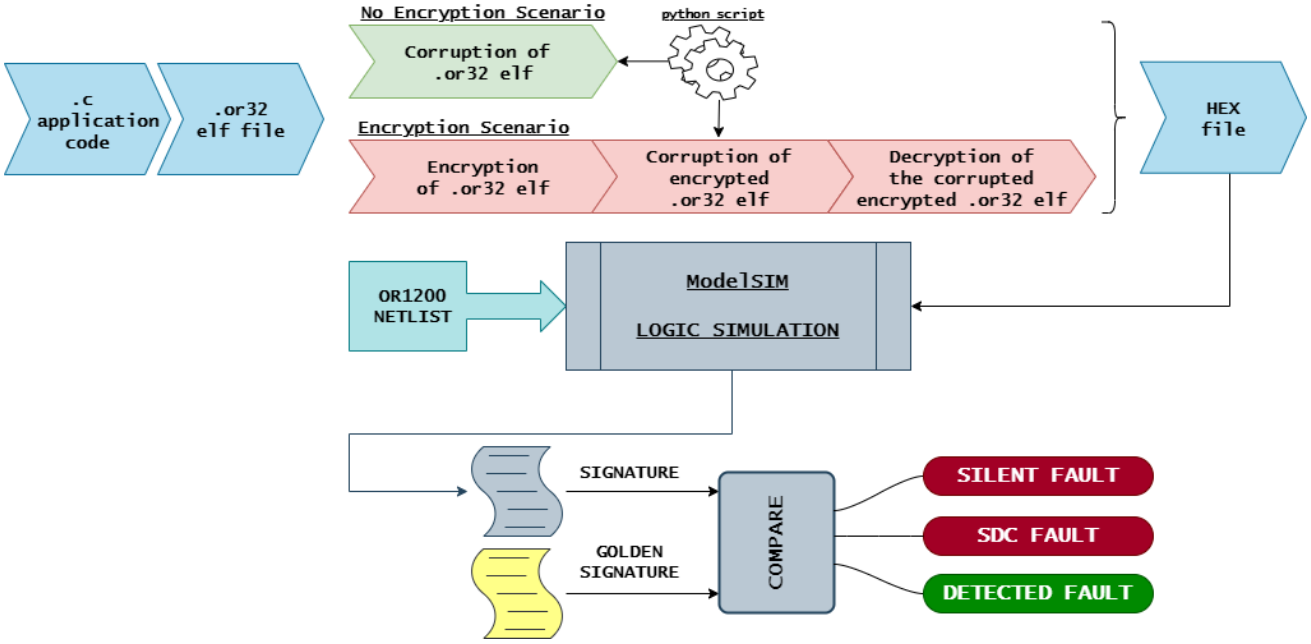

Fig. 2: Experiments' Setup

method is based on 128-bit blocks, *one bit flip on the memory results in at least four consecutive instructions that are entirely corrupted*. ECB, CBC and CFB modes of operations of block ciphers belong to this category. For the sake of clarity, in the following of the paper we will refer to these encryption methods with the generic name of *block ciphers*.

For these reasons, implementing memory encryption based on a propagating configuration may lead to an increased reliability of the system, substantially reducing the risk of SDC. In the next Section, we will verify these hypotheses through some experiments.

IV. Experimental Results

A. Experimental setup and environment

For the purposes of our experiments, we selected a set of programs representative of typical application codes. Matrix multiplication, the Dijkstra algorithm and the Bubble-sort algorithm were considered. These programs were written using ANSI C, then compiled for the OR1200 processor, providing us with the $.or32$ binary files. The OR1200 is a 32-bit scalar RISC with Harvard micro-architecture and 5 stage integer pipeline. The OR1200 core is mainly intended for embedded, portable and networking applications. The RTL description of the core was taken from [5] and it was synthesized using the Silvaco 45nm Open Cell library [10].

To analyze the effects of faults affecting the code memory content, we corrupted the binary files with a Python script. Given the binary file, the encryption method to be used and the number of faults to inject, the script (i) performs the encryption, (ii) injects the specified amount of faults, and (iii) performs the decryption. As a result, the corrupted version of the binary file is obtained. For the exhaustive 1-bit injection experiments we used the OpenSSL library [11] for encrypting/decrypting. Finally, the corrupted binary format was transformed into a hex-file for the purpose of the Logic Simulation with ModelSim, which follows.

To know when to set a timeout for every test program's simulation, we simulated the OR1200 with the original non-corrupted binary files and so we defined a timeout margin as:

$$T_{timeout} = T_i + \Delta \tag{1}$$

- $i \in \{MatrixMultiplication, Dijkstra, BubbleSort\}$.
- Δ is a time constant: $\Delta = 10\% \times T_i$.

After finding the timeout values and furthermore acquiring the golden values for every program, we run the Logic Simulation with the corrupted binary files. After the simulation end, each fault is classified as:

- *Silent*: the signatures are the same (effect-less fault).
- *Silent Data Corruption*: the signatures differ.
- *Detected*: the application crashed or the timeout condition was triggered.

The above described process (depicted in Figure 2) was fully automated using two bash scripts. The first script was responsible for corrupting the $.or32$ file, running the simulation and comparing the results with the golden values. The second script was used to run multiple experiments while changing the experiment's variables (e.g., the number of faults to inject).

B. Results and discussion

A set of experiments was devised and performed on the OR1200 microprocessor resorting to the described environment and procedure. To have a point of reference, we started by doing single-bit fault injections without using encryption.

TABLE I: Detection Rates on 1-bit Injections

Programs	Encryption Methods						
		Stream Ciphers			*Block Ciphers*		
	NO ENC	CTR	OFB	RC4	CBC	ECB	CFB
Matrix Multiplication	20.2%	22%	22.2%	21.8%	90 %	89.9%	84.1%
Bubble Sort	35.1%	34.6%	34.4%	34.2%	85.1%	85 %	81.5%
Dijkstra	34 %	33%	35%	35,5%	93.5%	93.5%	89.5%
Average	**29.8%**	**29.9%**	**30.5%**	**30.5%**	**89.5%**	**89.5%**	**85.0%**

	NO ENC	*Stream Ciphers*	*Block Ciphers*
Average detection	29.8%	30.3%	88.0%
Average Improvements*	**0% (ref.)**	**0.5%**	**58.2%**

*Average absolute improvements compared to unencrypted code

For the Dijkstra program, we injected 200 one-bit random faults, since we could not adopt the exhaustive injection due to the large application code size, which resulted in excessively long simulation times. Hence, we reduced the number of iterations to 200 since they were enough to provide accurate samples. For all the test programs we also performed one-bit fault injections for all the encryption methods in our set. The results of these experiments are shown in Table I. The table reports the detection rates for single-bit fault injections obtained by applying the considered proposed approach. The obtained results show that, even for single-bit error scenarios, block cipher encryption mechanisms provide a drastic detection increment (58.2% on average) compared to unencrypted code. Values spanning from 81.5% to 93.5% single bit-flip detection were achieved. The single-bit error event is the most difficult to detect via encryption/decryption. Indeed, this condition quite often leads to corrupted data (SDC), silently propagating the error to the rest of the computation.

```
1: for i = {MatrixMult, BubbleSort, Dijkstra} do
2:   for j = {CTR, OFB, CBC, ECB, CFB, RC4} do
3:     for k = {10, 20, 50, 100, 200, 500} do
4:       for (l = 1; l ≤ 1000; l++) do
5:         Simulate the injection of k faults using j on i;
6:         Classify each fault;
7:       end for
8:     end for
9:   end for
10: end for
```

Fig. 3: Random Fault Injection Procedure

The next step was to understand the ability of encryption to detect multiple-bit faults. For this purpose we computed, for each encryption mechanism, how many bit should be flipped before the number of SDC faults is reduced to 0. We started by injecting a small number of bit-flips so that we could distinguish the difference between the detection rates of stream ciphers and block ciphers. Then we moved to an increasingly large number of bit-flips. Lastly, we considered as many samples as possible from every method and every type of injection. Considering also the computational time, the complexity of the simulations and also based on some experiments, we selected a number of iterations for each experiment equal to 1,000. The procedure is described by the pseudo-code in Figure 3.

The graphs in Figures 4 to 6 show that using block ciphers over stream ciphers provides significantly higher detection rates for multiple-bit fault injection as well. These graphs also show that encryption alone is able to guarantee the detection of faults starting from a given multiplicity k. The value of k changes depending on the encryption method. In some cases, encryption is able to detect any fault with multiplicity greater than 4 or 5 bits. This is a remarkable result, since error correction codes are typically unable to guarantee similar detection capabilities.

V. Conclusions

Embedded systems are increasingly employed in safety-critical scenarios, both in industrial and in consumer environments. As a consequence, robustness and confidentiality of the application code running on those systems are key requirements. This generally results in implementing protection mechanisms for the memory containing the application code, in order to prevent both random error effect propagation (e.g., due to radiations) and unauthorized accesses. Encryption mechanisms are widely adopted to protect all the intellectual properties within the system. At the same time, designers often resort to expensive error detection/correction mechanisms to prevent random error effects that would cause unexpected results. In general, it is crucial to avoid the silent propagation of random fault effects, either by correcting the error or by preventing it from silently propagate.

In this work, we investigate the possibility to exploit the features of some encryption mechanisms to detect errors in the system: when a random fault strikes and impacts one or more bits of the encrypted code in the memory, the error will propagate to other bits of the code thanks to the decryption process. This will likely cause instruction corruptions, thus it will prevent the error from silently propagate.

Experimental results showed that, by using block cipher encryption mechanisms, the detection rate of single-bit faults increases on average by 58.2%, compared to non-encrypted code, without resorting to any ECC. In absolute terms, single bit-flip fault detection values, achieved with block cipher

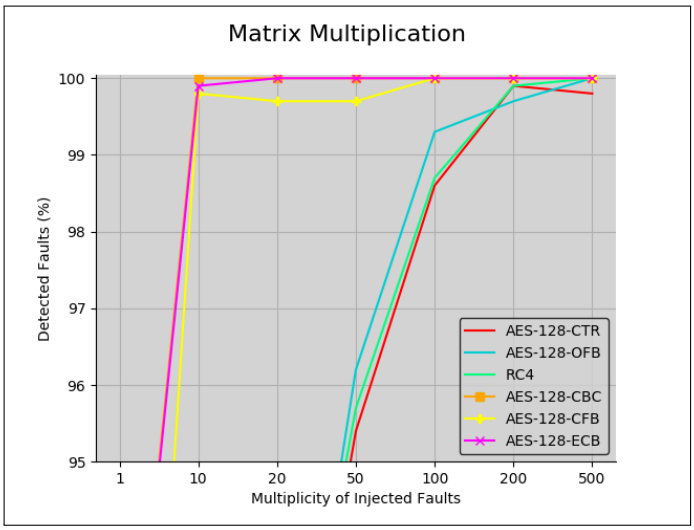

Fig. 4: Detection Rates on Matrix Multiplication

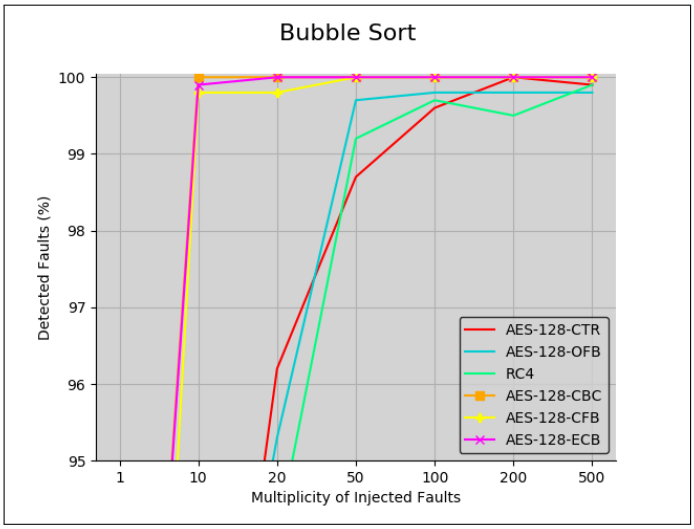

Fig. 5: Detection Rates on Bubble Sort

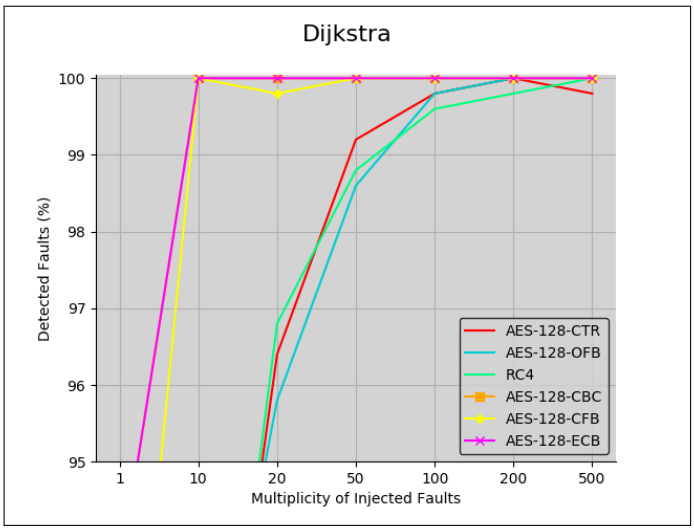

Fig. 6: Detection Rates on Dijkstra

encryption, spans from 81.5% to 93.5% (88.0% on average). These results support the claim that it is possible to avoid the implementation of costly detection/correction mechanisms by reusing encryption mechanisms already implemented to protect the system from illegal accesses. Moreover, some encryption/decryption mechanisms proved to have very good detection properties with respect to multiple-bit faults, thus complementing the limited capabilities of ECCs in this direction.

Work is currently being done to extend the performed work considering more application programs and different types of encryption mechanisms.

VI. Acknowledgments

Part of this work has been supported by the Center for Automotive Research and Sustainable mobility@PoliTO (CARS).

References

[1] Jun Yang, Lan Gao, and Youtao Zhang, "Improving memory encryption performance in secure processors," *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 630–640, May 2005.

[2] M. Ye, K. Zubair, A. Mohaisen, and A. Awad, "Towards low-cost mechanisms to enable restoration of encrypted non-volatile memories," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2019.

[3] J. Xiao-bo, T. Xue-ging, and H. Wei-pei, "Novel ecc structure and evaluation method for nand flash memory," *IEEE International System-on-Chip Conference*, 2015.

[4] J. A. Martínez, J. A. Maestro, and P. Reviriego, "A scheme to improve the intrinsic error detection of the instruction set architecture," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 103–106, July 2017.

[5] O. Community, "OpenRISC," https://openrisc.io, 2019, [Online; accessed 29-November-2019].

[6] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[7] E. Valea, M. Da Silva, M. Flottes, G. Di Natale, S. Dupuis, and B. Rouzeyre, "Providing confidentiality and integrity in ultra low power iot devices," in *2019 14th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*, April 2019, pp. 1–6.

[8] E. Valea, M. Da Silva, G. Di Natale, M. Flottes, and B. Rouzeyre, "A survey on security threats and countermeasures in ieee test standards," *IEEE Design & Test*, vol. 36, no. 3, pp. 95–116, June 2019.

[9] S. Kannan, N. Karimi, O. Sinanoglu, and R. Karri, "Security vulnerabilities of emerging nonvolatile main memories and countermeasures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 1, pp. 2–15, Jan 2015.

[10] Silvaco, "45nm Open Cell Library," https://www.silvaco.com.

[11] The OpenSSL Project, "OpenSSL Software Library," https://www.openssl.org.