UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Hardware-Trojan resilient blockcipher implementation based on multi-party computation

Dissertation presented by
Olivier BRONCHAIN, Louis DASSY

for obtaining the Master's degree in
Electrical Engineering

Supervisor
François-Xavier STANDAERT

Readers
Jean-Didier LEGAT, Loïc VAN OLDENEEL

Academic year 2016 – 2017

Abstract

Integrated circuits are now deployed in a continuously increasing range of applications. For many of them, sensitive data is manipulated, leading to technical and legal issues regarding its security. In this context, protection mechanisms are usually included in order to prevent various types of adversaries. At the most abstract level, cryptographic algorithms and protocols ensure that it is theoretically feasible to communicate privately and to guarantee message authenticity. More practically, various type of physical attacks can take advantage of features and imperfections in cryptographic implementations. Well-known threats are the cases of side-channel adversaries (taking advantage of unintended information leakages, e.g. due to the power consumption of the implementations) or fault adversaries (trying to force the implementation to perform erroneous computations). In this work, we are concerned with an even more powerful type of physical adversary, next denoted as the hardware Trojan adversary.

In summary, the hardware Trojan adversary is not only able to observe the implementation at run time, but to maliciously modify its hardware at manufacturing time. Typical examples of hardware Trojans are cheat codes (e.g., sending the secret data under some rare input pattern) or time bombs (e.g., sending secret data at some time). Such extreme adversaries are motivated by the increasing need of trust in integrated circuits. That is, recent news have shown that untrusted software is deployed and exploited (as typically emphasized by the Snowden revelations). Ultimately, this implies that the design of secure systems has to start by trusted hardware – a problem for which little solutions exist so far.

More precisely, the state-of-the-art literature suggests that detecting hardware Trojans is both technically challenging (if not impossible), and hard to formalize (which implies hard to quantify risks). Hence, an alternative is to prevent hardware Trojans actively. In this respect, a recent work published at ACM CCS 2016 introduced a theoretically founded way to mitigate hardware Trojans thanks to testing amplification. It essentially exploits secret sharing and multiparty computation to prevent cheat codes, and redundant randomized testing to prevent time bombs. Based on this solution, it is possible to render the probability of a hardware Trojan attack exponentially small, if the run time of the circuit is limited.

In this paper, we extend this theoretical work towards practice in two important directions. First, we designed a hardware architecture for a Trojan-resilient circuit for two block ciphers (the standard AES and lightweight Mysterion), based on an improved multi-party computation protocol, and implemented the architecture on a concrete prototype connecting three FPGAs. This allowed us to evaluate the performances of such a Trojan-resilient circuit on a concrete basis, confirm practical applicability, and to identify bottlenecks and tracks for improvement. Second, one core assumption of such Trojan-resilient circuits is the existence of a small trusted master circuit, of which the size has to be minimized. We analyzed the implementation of such a master, confirmed that it is indeed minimum compared to the implementation of the full block ciphers, confirming theoretical analyses with quantitative experimental data. We additionally investigated the effectiveness of a side-channel based hardware Trojan detection for such a small master and show positive results for the practically-relevant case of time bombs.

Acknowledgments

First and foremost, we would like to express our gratitude to our advisor Prof. Standaert for his advice, support, and continuous availability. He gave us complete intellectual freedom in our work, while always having valuable hindsight to offer.

Sincere thanks to Loïc for sharing his electronics prototyping and PCB design expertise, and for always stimulating our curiosity, even in only tangentially related areas. We were fortunate to be able to rely on him. His patience, availability, experience and generosity were a great help.

We would like to thank Anthony for his help, code and figures related to his work on XLS ciphers.

Thanks to Souley for his presence, good humor and help in the lab.

Finally, we would like to thank our friends and labmates for their hindsights and discussions, their LaTeX support, the late working nights company, and especially the good time spent on something other than our thesis.

Contents

Symbols

$\boldsymbol{C}$	The number of cycles taken to run a block cipher in a MPC protocol; C_t is the equivalent in time domain.
$\boldsymbol{Con}$	The number of cycles involved in control for a MPC block cipher implementation; Con_t is the equivalent in time domain.
$\boldsymbol{F_k}$	Is a keyed pseudo random permutation (block cipher) and F'_k denotes keyed correlated pseudo random permutation.
$\boldsymbol{L}$	The number of cycles involved in linear operation for a MPC block cipher implementation; L_t is the equivalent in time domain.
$\boldsymbol{L_{IC}}$	Total leakage of an integrated circuit (power consumption) as a function of time, including noise.
$\boldsymbol{T}$	The number of cycles involved in data transfer in a MPC block cipher implementation; T_t is the equivalent in time domain. Can also denote the tester for HT resilient architecture.
$\boldsymbol{TR}$	Compilation of a given circuit specification for HT resilient architecture.
$\boldsymbol{\Delta}$	Difference of two signals in the Fourier domain.
$\boldsymbol{\Gamma}$	Specification that a given integrated circuit should follow. This is equivalent to the golden netlist.
$\mathbf{ROB}_{\Pi}$	Robustness game representing the security game for HT resilient architecture.
$\boldsymbol{\alpha}$	Algorithm that is performed following a given MPC protocol, used to define intrinsic properties of given algorithm.
$\boldsymbol{\beta}$	The bus quality factor in $[bit/cycle]$ represents the amount of bits that can be sent/received via a bus in terms of internal cycles. β' denotes the same quantity normalized by the bus width.
$\boldsymbol{\beta_{IC}}$	The leakage of the gates of a given integrated circuit depending of its inputs and outputs as a function of time.
$\boldsymbol{\delta}$	Difference of two signals in the time domain or isomorphic transformation $GF\left(2^8\right) \mapsto GF\left((2^4)^2\right)$.
$\boldsymbol{\epsilon}$	Denotes the probability for an HT to trigger.
$\boldsymbol{\eta}$	Additive withe Gaussian noise defined by its mean and variance. It is stationary.
$\odot$	Bitwise AND or field multiplication depending on the input elements.
$\boldsymbol{\lambda}$	Number of triplet performing independently the MPC protocol used in the Hardware-Trojan resilient architecture.
$\boldsymbol{\mathcal{A}}$	A PPT attacker that is building untrusted devices and can choose the input of the architecture build with those ICs.
$\boldsymbol{\mathcal{F}}$	Denotes the Fourier Transform of any signal (generally time-based signal).
$\boldsymbol{\mathcal{H}}$	Hamming weight, meaning the number of difference between two bit strings. Equivalent to $\oplus$ in case of a single bit.

τ_{IC}	The leakage of the control gates of a given integrated circuit as a function of time.
θ	Denotes a given implementation of an algorithm that is performed following a given MPC protocol.
$\oplus$	Bitwise XOR or field addition depending on the input elements.
e	Euler constant.
$f_{\mathbf{bus}}$	Bus clock frequency used to communicate between the slaves and the master.
$f_{\mathbf{int}}$	Frequency of the main internal clock of an electronic device.
n	Number of runs with guaranteed security bounds in the Hardware-Trojan resilient architecture.
t	Number of tests performed independently on slaves in the Hardware-Trojan resilient architecture.

Introduction

Motivation

The need in information security is certainly one of the most important topics in modern society. In 2013, Edward Snowden has shown to the world the NSA was collecting worldwide metadata regarding all internet users. After those revelations, the public has been aware that protection of private information is essential to preserve personal privacy. More recently, during the election in the U.S. or in France, private information leakage was able to influence the future of nations. In that context, security researchers and industries are developing solutions to reach better privacy.

In modern systems, hardware is the physical support for digital data, and this support has to be trusted. However, in different ways, this is not yet the case. Indeed, there is no efficient methodology to verify that a circuit is always operating honestly. In addition, some information can leak from a circuit due to the underlying physical effects that are actually supporting the data. A clever attacker could exploit this intrinsic behavior to achieve malicious operations.

The modern electronics industry is traditionally building solutions by taking into account the power, performances and area trade-offs. However, due to the increasing need in privacy, the security should also be part of this trade-off [37]. This work proposes a framework to prevent malicious modifications of a circuit, called Hardware-Trojan, using multi-party computation and develop the case study of cryptographic block ciphers.

Problem statement

Due to the globalization of integrated circuit market but also to the increasing complexity of digital systems, companies tends to subcontract parts of the fabrication process. For example, engineers include third-party intellectual properties (IP) in their designs to accelerate the development time, and send the circuit description to a remote foundry.

As shown in Figure A, this process can lead to the introduction of a Hardware-Trojan (HT) in the original design, which could alter the expected functionality of the circuit. For example, malicious behaviors of an HT could lead to the denial of service or private information leakage. In the example from Figure A, an engineer is creating a design that includes the IP as third-party module and he wants to put it into production. The first issue is that he has no guaranty that the IP he included is honest in the sense that it does not include unexpected hardware. The second weakness is that to create the desired system at a logic gate level, the engineer uses dedicated tools that maps a hardware description language (HDL) to the layout of physical logic gates, called a netlist. However, there is no guarantee that such tool is honest meaning that it does not add unwanted gates. To get a physical implementation of the system, the netlist is sent to a

remote foundry, which is untrusted. The manufactured chip could have been infected at three different steps: IP insertion, transformation by the design tools, and fabrication.

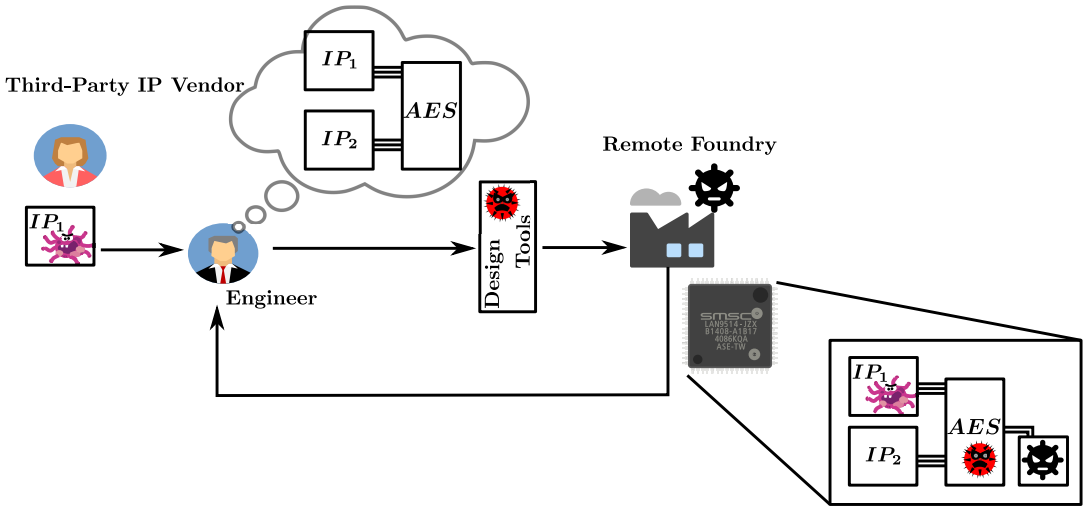


Figure A: Typical issue due to globalization of IC market.

A problem is so raised: *"how can an Hardware-Trojan be detected or evaded in an efficient manner ?"*. The ultimate goal is to provide formal quantitative proof of security against Hardware-Trojan attacks that would guarantee the reliability of physical devices.

State-of-the-art

By nature, a Hardware-Trojan must be furtive otherwise it would directly be detected by simple testing. In a general model, a HT is triggered before demonstrating any malicious behaviors. The HTs are represented in three main categories depending on the trigger type used.

1. *Time bomb*: a HT is triggered after a given amount of time. A typical example is the case of a counter which counts the number of time n the circuit was used. The malicious behavior is triggered when n becomes higher than a predetermined threshold.

2. *Cheat Code*: when the circuit reaches a given state, the HT is triggered. A typical example is tied to the input of the circuit: when it receives a given value at the input, the HT starts to manifest unexpected action.

3. *Physical*: due to a given external analog trigger, the HT is triggered if it is placed in given environment conditions such as specific temperature or position. This family also includes covert communication channels like radio frequency transmission.

The Hardware-Trojan detection methodologies are split into two main families as shown in Figure B. A first class of solutions is depicted as destructive, meaning that such a technique destroys the device under test to establish if it is trojan free. A traditional solution suggests reverse engineering an integrated circuit (IC) [19, 39] to determine whether or not it is honest. Other solutions propose to compare a candidate circuit by visual inspection with a trojan free (golden) model to determine if the circuit is infected [6]. The visual inspection is generally performed with a scanning electron microscope (SEM). In addition, by destructively verifying some samples in a given set of IC, there is no guarantee for the remaining circuits in the set [37]. For those reasons, destructive solutions are not considered as viable for very large scale integration (VLSI) since they are costly and do not ensure a given security level.

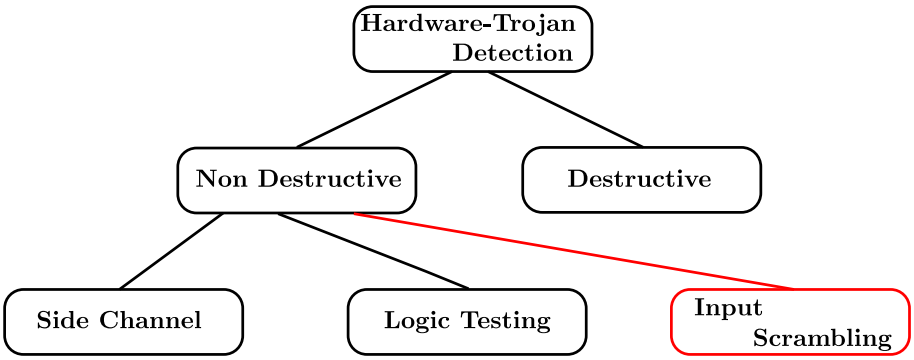


Figure B: Different Hardware-Trojan detection methods.

A natural alternative to destructive solutions is to detect the HT while leaving the device intact. Those solutions are called non-destructive. A first option is to try to spot the HT by looking at the differences between the side-channel leakage of a golden model and the one from the device under test. Side-channels are defined as all the physical information that can be extracted from a device and were first used to recover keys in cryptographic implementations [23]. In the field of Hardware-Trojan detection, it is pointed that side-channels can be used as the signature of a design: techniques that exploits power consumption variations [1] and electromagnetic emissions [30] to detect HT are often used by researchers.

The main drawbacks of those techniques is that they are sensitive to the variations associated with the manufacturing process that are increased with the reduction of the circuit feature size [33, 31]. In the case of an HT that are small compared to the original circuit, the side-channel variations introduced by the trojan could be lost in the process noise. Against that intrinsic physical limitation, researchers suggest to take into account multiple side-channel sources, for example power consumption and maximum frequency. With this solution, they are able to detect relatively small trojans even in presence of process variation [29]. It is important to note that those detection schemes require a golden model: [1] points out the possibility of using destructive detection to obtain a golden model, and is now a community consensus. Finally, just as in the the traditional side-channel community [10], the lack of formal security proofs makes us less confident in the security of those solutions.

Non-destructive detection is not limited to side-channel analysis: logic testing aims to trigger the Hardware-Trojan by applying a restricted number of inputs and observe corrupted outputs [5]. Different algorithms were found to verify that a netlist is Trojan free: FANCI [42], UCI [18] or VeriTrust [49]. Those three detected all HT from the TrustHub benchmark [38], but there were in practice targeting only a subset of all the possible HT. DeTrust [50] has exploited that weakness to construct a methodology to design an HT that escapes the previously mentioned detection algorithms. Last year, an efficient methodology to detect any HT included in a netlist in a polynomial time was suggested (HaTCh [16]), which defeats DeTrust by detecting all the HT it generates. Nevertheless, those methods are not able to capture HT exploiting physical effects like triggering using a rare critical path [12]. Those methods should so be restricted to IP honesty verification usage. For any type of logic testing, a Hardware-Trojan could target the testing phase [5] and get around any detection methodology.

Another promising solution is so called *"input scrambling"* [41]: by making the input of the circuit uniformly distributed, a potential Hardware-Trojan could not be triggered by a *"cheat-code"* selected by an attacker with a non negligible probability. However that solution is exploitable only in the case of *homomorphic* functions as i.e asymmetric cryptography algorithm RSA [41].

A recent work [11] published at ACM CCS 2016 introduced a theoretically founded way to mitigate hardware Trojans thanks to testing amplification. It essentially exploits secret sharing and multi-party computation to prevent cheat codes similarly to *"input scrambling"* , and redundant randomized testing to prevent time bombs. Based on this solution, it is possible to render the probability of a hardware Trojan attack exponentially small, if the run time of the circuit is limited.

That article raised two open questions. Is it really possible to perform cryptographic primitives in that framework with reasonable performances ? Does that construction really reduce the number of trusted logic gates compared to a classical implementation ? In the case of a drastically reduced number of genuine logic gates, this would reduce the confidence problem to a very small device that would enhance classical detection techniques performances.

Contributions

Some improvements to [11] were introduced for performances enhancement namely choosing another multi-party protocol to maximize the performances of the cryptographic primitives computed using MPC protocol. This step is achieved by introducing the recently published multi-party computation protocol which is characterized by its throughput efficiency [2]. In addition, efficient representation of block ciphers was reached based on the side channel community work that is targeting properties for cryptographic primitives efficient in a multi-party computation protocol.

Based on those modifications, a hardware architecture is exposed and was designed by keeping in mind underlying the cryptographic hypothesis. That architecture has confirmed that their practical implementation is feasibility. Thanks to behavioral simulations, the bottlenecks of the system were identified and its influence on the global performances is highlighted. A proof for concrete Hardware-Trojan resilient systems based on multi-party computation is established thanks to a custom demonstration board including four independent FGPAs. Thanks to its maximum throughput of 1.55 Mbps for AES and 3.1 Mbps for Mysterion block ciphers, its viability is demonstrated for various range of application going from low to high throughput requirement.

An important effort has been made to reduce the amount of trusted gates to reach a drastically smaller number compared to classical implementation. That point has a non negligible impact on classical side-channel detection techniques due to the increased trojan footprint compared to the genuine gates. This is highlighted especially for a new technique using those pre-existing tools applied in the frequency domain for a subspace of Hardware-Trojan namely *"time bomb"*.

This work is structured as follow: first, the background needed to understand the contribution is described. An algorithm to convert any circuit into a trojan resilient circuit is presented. This algorithm is then applied to AES and Mysterion block ciphers. An upper bound for the performance is defined. Then, a global hardware architecture is proposed and explained in a top down fashion. Simulations results are shown, and the performance trade off are discussed. The real physical system is described and the performances obtained on the demonstration board are presented. Finally, some solutions and experiments to verify a small size circuit are proposed.

1

Background

This section presents background information required to understand the contribution. In any section, a reader familiar with the subject at hand can skip over, as the information is self-contained.

1.1 Block ciphers

Block ciphers are one of most important building blocks in modern cryptography. They allow for secure communication between two parties using symmetric encryption, meaning that those two parties hold the same key k_s. Without that key k_s, it is not possible to recover the plaintext.

This section first defines formally a block cipher and presents its typical architecture. Secondly, two different block ciphers are described, namely AES and Mysterion.

1.1.1 Block cipher definition

As mentioned above, a block cipher can be used to secure a communication between two parties. A classical example is the one of two lovers Alice and Bob. They would like to communicate without being spied on by an adversary. In the example from Figure 1.1, Alice wants to send "Hello Bob" to her lover.

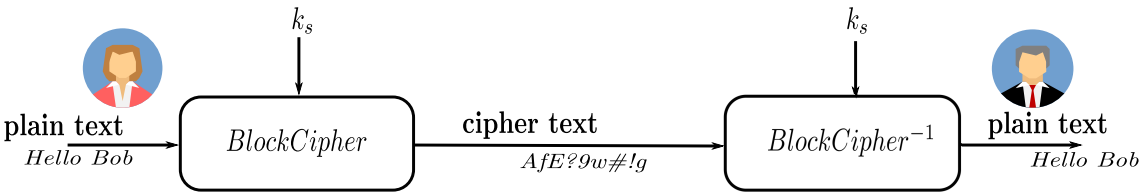


Figure 1.1: Encrypted communication between Alice (left) and Bob (right).

This message is called the plaintext. Before sending the message, she encrypts the plaintext using the block cipher and the key k_s that Bob already knows. The resulting data is called the ciphertext. Without knowing the key k_s, it is supposed to be impossible to retrieve the plaintext. So Alice can ask someone to give the ciphertext to Bob. Once Bob receives it, he can retrieve the plaintext by inverting the block cipher.

Pseudorandom permutation According to [21, p. 77-82], a block cipher $F_k(\cdot)$ is designed to be a secure instantiation of a pseudorandom permutations with a fixed key and block length.

Such a keyed permutation can be formally written as $F : \{0,1\}^n \times \{0,1\}^n \mapsto \{0,1\}^n$ and can be interpreted in the following way.

First, one may observe[1] there are 2^n possible $F_k : \{0,1\}^n \mapsto \{0,1\}^n$. By choosing a key k, one of those 2^n permutation is selected to perform the encryption. This is precisely the knowledge of that permutation that allows encrypting or decrypting data securely.

In addition, the set of the keyed permutations is called efficient if applying $F_k(x)$ or inverting $F_k^{-1}(x')$ can be done in polynomial time with respect to x and k, and if given k, that pseudo-random permutation cannot be distinguished from a random permutation in a polynomial time. Selecting a set of keyed permutation F corresponds to selecting a block cipher.

Typical structure A blockcipher must behave like a pseudorandom permutation. This implies that changing a single bit at the input of $F_k(\cdot)$ must result in a completely different output. To implement block ciphers, Shannon introduced the confusion-diffusion paradigm. A typical block cipher built on a substitution-permutation network (SPN) is built by repeating those three steps in multiple rounds [21, p. 204-206].

1. *Key mixing*: Set $x := x \oplus k$
2. *Confusion*: Set $x := f_0(x_0)||f_1(x_1)||\ldots||f_{15}(x_{15})$
3. *Diffusion*: Set $x := P(x)$, where P permutes the bits of x.

Confusion is the idea of building a permutation $F(\cdot)$ on long (i.e 128 bits) string, based on smaller known permutations $f(\cdot)$ (i.e 8 bits) providing highly non-linear behavior. Those functions $f(\cdot)$ are also called S-box. The output of $f(\cdot)$ can be concatenated as in Eq. 1.1 with $x_i \in \{0,1\}^8$ and $x \in \{0,1\}^{128}$. However, with such a permutation, changing a bit in x_0 does not change the entire output.

$$F(x) = F(x_0||x_1||\ldots||x_{15}) = f_0(x_0)||f_1(x_1)||\ldots||f_{15}(x_{15}) \tag{1.1}$$

This is why a *diffusion* layer is required. This act by mixing the bits at the output of the *confusion* layer to spread a change across the entire output. In addition to those two layers, a *key mixing* layer is introduced to get a keyed permutation.

Hereunder, two different block ciphers are introduced, namely the standard and worldwide used AES and the lightweight Mysterion, optimized for masked operation which is therefore an advantage in multi-party computation.

1.1.2 AES building blocks

In 1997, the `National Institute of Standards and Technology` (NIST) started the `Advanced Encryption Standard` (AES) process in order to standardize on a block cipher to succeed to DES. The Rijndael algorithm was first published in 1998 and was adopted as AES in 2001 [8]. Nowadays, the Rijndael algorithm is used worldwide. `AES` is a block cipher operating with key sizes of 128, 196 or 256 bits. In this work, the focus will be on the 128 bits version.

Rijndael implements Shannon's confusion and diffusion principle [36] based on several building blocks that are repeated over 10 rounds[2]. *MixColumns* and *ShiftRows* operations are responsible for the diffusion layer while *SubByte* is responsible for the confusion layer. The *AddRoundKey* ensures that having the key k_i is required to reverse the block cipher operation. The key k_i depends on the round i and is derived by a key expansion algorithm based on the input key.

[1] The set of all the possible permutation Perm_n is of size $(2^n)!$ while the set of all the keyed permutation is of size 2^n

[2] The number of rounds depends on the key size. Here the case of 128 bits AES is assumed

Figure 1.2 describes the block diagram of an AES encryption.

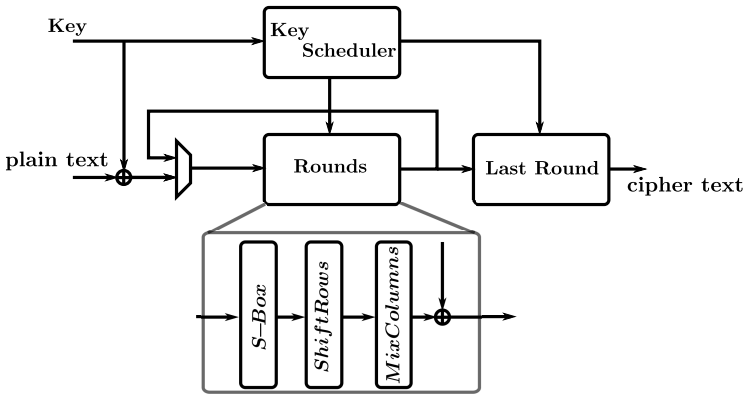


Figure 1.2: AES building blocks.

The building blocks of the AES are based on arithmetic in $GF(2^8)$ as described in [8]. They are described below.

Add Round Key The Add Round Key is a bitwise XOR between the state and the round key. It is important to note that the round key is derived based on the cipher key.

ShiftRows ShiftRows is a simple wire crossing. It exchanges the bytes as described in Figure 1.3. This operation can trivially be inverted. It is also part of the linear layer of the Rijndael block cipher that ensures diffusion.

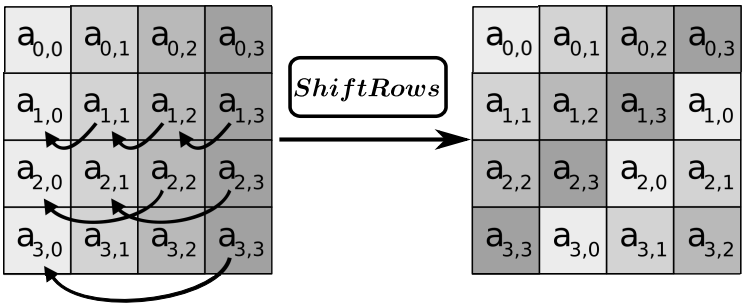


Figure 1.3: Shift Rows

MixColumns MixColumns is also part of the linear layer of the Rijndael block cipher. MixColumns consists of a four terms polynomial $b(x) = b_3x^3 + b_2x^2 + b_1x + b_0$ with b_i a byte of the AES state and is therefore an field element of $GF(2^8)$. That polynomial is multiplied by a constant $a(x) = 03\ x^3 + 01\ x^2 + 01\ x + 02$. As mentioned in [8], that multiplication can be seen as a multiplication with a circulant matrix. The MixColumns transformation can be then written as:

$$\begin{bmatrix} b0 \\ b1 \\ b2 \\ b3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a0 \\ a1 \\ a2 \\ a3 \end{bmatrix}$$

SubBytes SubBytes is the only non-linear block of the AES. It consists of an inversion in $GF(2^8)$ followed by an affine transformation, and is performed on each byte independently. The inversion is non-linear. The resulting value for a byte a is given by:

$$b = \text{affine}(a^{-1})$$

where the affine transformation is given by a matrix multiplication modulo 2. The matrix for the affine transformation is:

$$\begin{bmatrix} y0 \\ y1 \\ y2 \\ y3 \\ y4 \\ y5 \\ y6 \\ y7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \odot \begin{bmatrix} x0 \\ x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

For the rest of this work, the official *SubByte* name of the AES *confusion* is refered using the term S-box for consistency with Mysterion and typical cryptographic notation.

1.1.3 Generic bitslice block cipher

A bitslice cipher is a block cipher constructed only from bitslicing operations, i.e. bitwise operator applied across a block of bits. The main difference with a block cipher like AES is that the operation are no more performed on bytes.

LS-designs block ciphers, introduced in [13], are bitslice ciphers aiming to be efficiently implemented in a masked setting. Considering a block cipher as a succession of linear and non-linear bitslice operations namely L-box and S-box as shown in Algorithm 1. Generally the L-box is based on look-up-tables while the S-box is designed to be efficient in masked setting by minimizing its internal non linear logic.

LS-designs naturally shift the block cipher operation balance towards more linear ones, since those will not require special treatment in a masked implementation, on the contrary of non-linear operations.

Algorithm 1: Generic LS-Design [20]

```
1 X ← P ⊕ K
2 for 0 ≤ r < N_r do
3 |   X ← Sbox(X) ;                                  // S-box
  |   X ← Lbox(X) ;                                  // L-Box
  |   X ← X ⊕ K ⊕ C(r) ;          // Round constant and Key addition
```

1.1.4 Mysterion building blocks

Mysterion [20] is a lightweight bitslice cipher. The origins of Mysterion lies in an extension of the LS-designs, aptly named eXtended LS-design (XLS-design). LS-designs suffer from a design flaw making them vulnerable to invariant subspace attacks [25]. In a nutshell, invariant subspace attacks, first introduced in [24], use a weak subset of keys to trigger an affine mapping between the round input and output. Since LS-designs always use the master key, and since the mixing operations only mix on rows, the invariant subspace property holding for one round will hold

for the next, and the whole cipher is weak if weak round constants are used. XLS-design, shown in Algorithm 2, adds one step compared to Algorithm 1.

Algorithm 2: Generic XLS-Design [20]

1 $X \leftarrow P \oplus K$
2 **for** $0 \leq r < N_r$ **do**
3 $\quad X \leftarrow Sbox(X)$; `// S-box`
4 $\quad X \leftarrow Lbox(X)$; `// L-Box`
5 $\quad X \leftarrow ShiftColumns(X)$; `// ShiftColumns`
6 $\quad X \leftarrow X \oplus K \oplus C(r)$; `// Round constant and Key addition`

The XLS structure of Mysterion is not vulnerable to invariant subspace attacks. The structure consists of first a key addition, then a non-linear substitution, the S-box, then a linear diffusion, the L-box , and finally a mixing operations, ShiftColumns, computed on the columns. Compared to a LS-design, the L-box is modified and the ShiftColumns operation, similar to the MixColumns operation of AES, is added. Those improve the security against invariant subspace attacks. A detailed view of the S-box, L-box and ShiftColumns is shown in Figure 1.4.

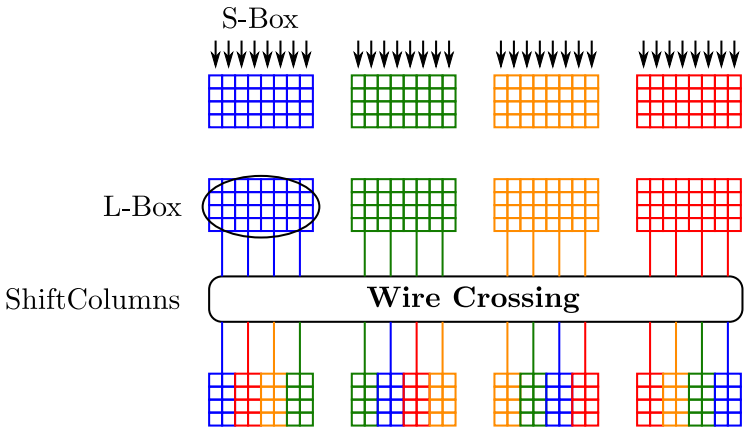


Figure 1.4: Mysterion S-box, L-box and ShiftColumns [20].

Compared to AES, Mysterion is simpler: there is no key expansion and the design of the S-box is more straightforward, since it only uses bitslice operations. It does not require inversion in $GF\left(2^8\right)$. The linear part presents some similarities with AES, since it also uses ShiftColumns which is equivalent to MixColumns in AES. In addition, Mysterion requires 12 rounds while AES only 10. Each step is detailed hereunder.

Key addition As in AES, this step consists in a bitwise XOR between the cipher state and the key.

S-box The non-linear part of the cipher (*confusion*), the S-box, is a Class-13 S-box [40]: it works with 4 bits at the time, and is optimal against linear and differential cryptanalysis [17]. Furthermore, it only requires to perform 4 AND operations. A full round will thus require 128 AND, those being the only operations requiring special treatment in a multi-party implementation.

L-box The *diffusion* provided by the L-box consists in a Maximum Distance Separable (MDS) code. MDS codes have the largest possible minimal distance between the input and output bits,

and thus provides optimal linear diffusion [3]. The L-box is performed in a bitslice fashion, on 32 bits blocks.

ShiftColumns As shown in Figure 1.4, the ShiftColumns operation mixes together the columns of the state, two by two. It provides *diffusion* after the 32 bits S-box and L-box layers. The ShiftColumns operation can be implemented efficiently using bitwise operations on a 32-bit word.

1.1.5 Multi-party Computation

Before introducing multi-party computation, secret sharing must be explained. Secret sharing allows to distribute among a group of N parties a secret value v with each party holding a value x_i. A share x_i is not useful on its own since it does not reveal any information about the shared value v. Depending on the secret sharing used, the amount of shares M needed to reconstruct v along the N possible may vary. A secret sharing is called an M-out-of-N sharing if M shares are needed to reconstruct the value v shared between N parties.

A simple example is the following two-out-of-two secret sharing of v. The two shares are r and $x = v \oplus r$, with r a uniformly distributed value. The shares r and x are randomly distributed and therefore do not give any information about v when observed independently. However, the shared value can be retrieved by using those two shares and computing $r \oplus x = v$.

Multi-party computation is a cryptography sub-field, where multiple parties want to perform some computation while preserving the privacy of their inputs using secret sharing. More formally, the parties $\{p_1, p_2, ..., p_n\}$ hold respectively the private data $\{d_1, d_2, ..., d_n\}$ and would like to learn the outcome y of a function $F : \{d\}^n \rightarrow y$. A multi-party computation protocol allows computing $F(d_1, d_2, ..., d_n)$ without revealing the individual d_i to the other parties [7].

Computing this function F implies to perform additions and multiplications on that secret sharing while keeping the shared value secret. The addition is linear and does not require specific data exchange, while multiplication, which are non-linear, will require to exchange bits between the different parties. The multi-party computation protocols try to perform those operations with a sufficient security while minimizing the exchange between the parties and the storage requirements.

Multi-party computation protocols can be used in the case of trojan resilient cryptography to mitigate "cheat code" as exploited by [11]. In this work, it is suggested to use the multi-party computation protocol described in section 2.2 [2].

1.2 FPGA

An FPGA (Field Programmable Gate Array) is an integrated circuit that can be programmed multiple times after manufacturing. It is configured using a hardware description language (HDL) like Verilog or VHDL. The synthesis tools map the programmer intended circuit to the actual internal elements of the FPGA, and the I/O of this circuit to the package pins.

1.2.1 Typical logic block

The logic blocks of the FPGA emulate the logic gates of the programmer's circuit. A simplified view of the logic block of a Spartan 6 FPGA [45, pp. 13] is given in Figure 1.5. It consists of look-up tables (LUT) and D-Flip-Flop, and implements the combinatorial and sequential logic operations described in HDL. Those logic blocks are connected together using configurable routing logic, as shown in Figure 1.6. The routing logic can also connect the digital signals to purpose-built circuit, for example differential pair drivers for the I/O or fast hardware multipliers.

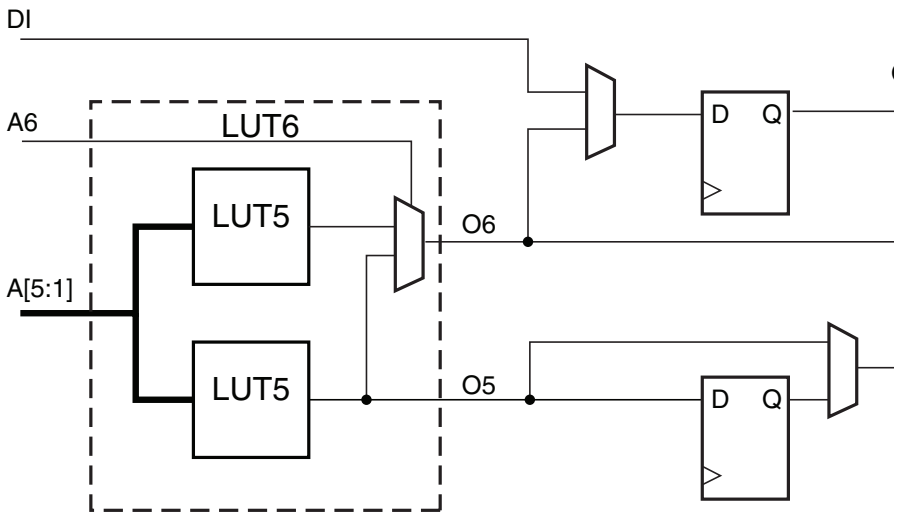


Figure 1.5: Spartan 6 simplified logic block, with two 5 bits look-up tables, two D-FF with one that can be bypassed to provide combinatorial logic.

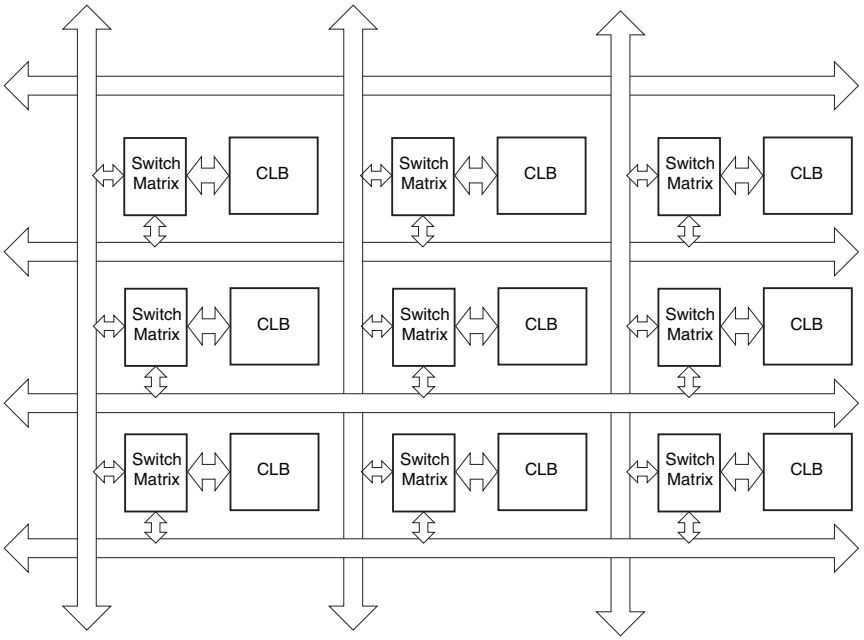


Figure 1.6: Reconfigurable routing between different logic blocks. From [45, pp. 37].

2

Algorithmic Improved Hardware-Trojan Resilient Block Cipher Implementation

Building an efficient Hardware-Trojan resilient block cipher starts with efficient algorithms. In that goal, a theoretical framework is exposed defining security model and security bounds for the implemented cryptographic algorithms. Secondly, since the proposed framework is based on multi-party computation, an improvement at that level is proposed. It allows to significantly increase the performances. Finally, performances in terms of exchanged bits are highlighted for different block ciphers namely AES and Mysterion. In the case of AES, two possible implementations are suggested, with one being significantly better than the naive version.

2.1 Trojan resilient cryptography

As mentioned earlier, Hardware-Trojans are an issue in modern integrated circuit: can we be sure that an IC does not contain malicious hardware? This section describes the state-of-the-art technique to implement Trojan resilient IC using multi-party computation [11], and the associated challenges. This solution is based on *input scrambling* [41] using a multi-party computation protocol, and testing.

2.1.1 General principle

The trojan resilient architecture from [11] relies on the transformation of an initial circuit and a testing phase. The whole process is defined as follows:

1. *Circuit compilation*: mapping the original circuit specifications to a triplet of "slave" sub-circuits (Figure 2.1) and a "master" circuit. The slave triplet will be repeated λ times and perform the original circuit specifications using an MPC protocol. Those triplets are composed of slave devices D^i build by a malicious manufacturer $\mathcal{A}$ that may not follow its specification Γ^i. Note that the slaves are untrusted and built by $\mathcal{A}$, while the master must be trusted and independent of the slaves complexity.

2. *Testing*: performing a PPT number of test t on each sub-circuit D^i independently. Those tests allow to verify that the device D^i implements correctly Γ^i during that testing phase.

3. *Real runs*: after connecting all the devices D^i to build the triplets, running at most n times the architecture that implements the original circuit specification. This ensure that the probability of $\mathcal{A}$ is performing malicious operations on the architecture is bounded to $(\frac{n}{t})^{\frac{\lambda}{2}}$.

The building blocks of this Trojan resilient architecture are explained hereunder, starting with the attacker model, then the three different steps, and finally the security guarantees.

2.1.1.1 Attacker model

For Hardware-Trojan, the attacker $\mathcal{A}$ is a malicious manufacturer. He receives the specification Γ^i of a circuit, and outputs a device D^i that supposedly implements the functionality of Γ^i. It is assumed that the attacker is PPT, i.e. has a bounded computing power. Note that the only assumption made on $\mathcal{A}$ is that it is PPT[1]; no assumption is made on the Trojan.

In the proposed protection scheme, the malicious manufacturer $\mathcal{A}$ is only responsible for building the untrusted slaves, and the master, which must be trusted, is built by an honest manufacturer. This is similar to split manufacturing as introduced in [4].

2.1.1.2 Circuit compilation

To obtain a hardware trojan resilient circuit [11], the original circuit specification Γ is transformed to a multi-party scheme Γ': one small and trusted master circuit communicates with three untrusted, black box slaves. Formally, the compilation process can be written as $\Gamma' \leftarrow \mathrm{TR}(1^k, \Gamma)$, where k is the security parameter of the PRNG supporting the multi-party scheme, and Γ' is the description of the master and the three slaves. The split scheme is repeated λ times, with λ different slave triplets running λ independent instances of the secret sharing, to enhance the security guarantees.

The result of that compilation is shown in Figure 2.1: one master is connected to λ triplets. Each of those is composed of three slaves S_j^i with i the triplet number and j its identity inside the triplet. Note that all the slaves are connected to the master, but they are not directly interconnected. Therefore, all of their communication happens through the master.

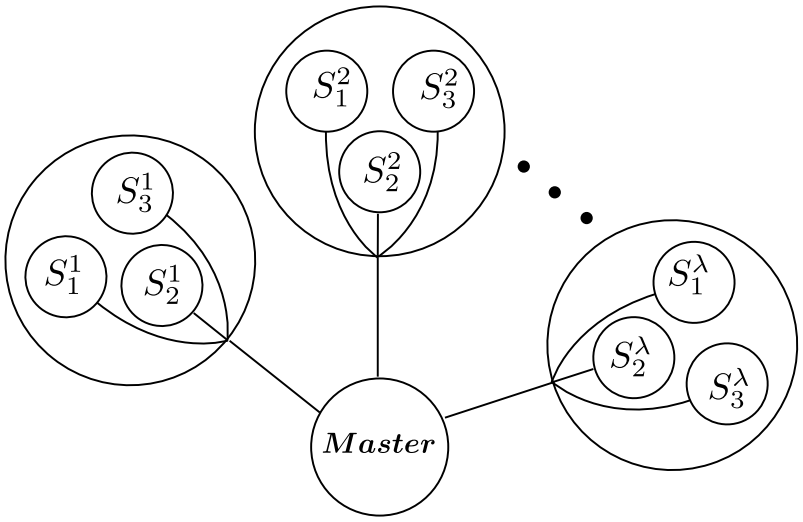


Figure 2.1: Sub-circuit splitting of [11], with λ triplets of sub-circuits. Each triplet i has three slaves denoted S_1^i, S_2^i and S_3^i.

To evaluate the function of the original circuit specification Γ, the architecture from Figure 2.1 performs the multi-party computation protocol described in section 2.2. The data that needs to be processed to carry out the original operation is secret-shared by the master, i.e. it is scrambled such that the data seen by one slave is uniformly random. The final output of the scheme consist

[1] Otherwise, $\mathcal{A}$ could break the computationally secure secret sharing.

of a reconstruction of the the value held by the λ instances, then a majority vote between them.

It is important to note that any multi-party computation protocol can be used. Indeed, the slaves are following exactly the MPC protocol up to a negligible probability, since this is verified during the testing phase. In addition, the slaves cannot communicate between them to reconstruct the secret since they are not physically connected[1]. Therefore, the MPC protocol chosen in section 2.2 is semi-honest since it usually allows for better performances. The drawback of a semi-honest protocol is the weaker security: if the slave can deviate from the protocol, the security is compromised. Thanks to the testing phase, this is only possible up to a negligible probability, and the weaker security is not an issue.

2.1.1.3 Test amplification

This phase consists in testing each sub-circuit up to t times. This allows the system to be resilient to two types of trigger: cheat-codes and time bombs.

The input scrambling performed in this circuit-splitting scheme avoids setting off "cheat-codes", i.e. special inputs that would trigger a malicious behavior in the slave circuit. Indeed, when using a multi-party computation protocol, the input of the slaves is uniformly random, such that no slave will learn anything about the original circuit input. With uniformly random input, either the malicious behavior can be caught during testing, or it will have a negligible probability of manifesting itself during the real runs. This probability is low as long as the number of tests t' performed is reasonably high compared to the number of real runs n, i.e. $t' \gg n$. This method is thus only valid when n is bounded.

Like cheat-codes, avoiding "time bombs" triggers of the trojan relies on testing. Assuming the testing is performed t times, if the malicious behavior only triggers after $t+1$ executions, then it would not be uncovered. The solution to this problem is to test t' times, with t' randomly selected: $t' \leftarrow \{1, \ldots, t\}$.

Finally, since the tester must run in PPT, the slave circuit must be verifiable in PPT, and can not contain a TRNG. This forces the MPC scheme to use a PRNG for the correlated randomness generation, which is only computationally secure.

2.1.1.4 Security guarantees

The following paragraphs describes the security guarantees of the theoretical framework from [11]. The full details and proofs are available in the original paper [11].

Security framework The first step when defining guarantees is to define a security game, namely for robustness: here the $\mathsf{ROB}_{\Pi}(\mathcal{A}, \mathsf{pub}, \Gamma)$ is defined and exposed in Figure 2.2[2]. It takes as input an attacker $\mathcal{A}$, the public parameters pub and a circuit specification Γ. The game ROB_{Π} returns 1 if the attacker $\mathcal{A}$ wins the game, i.e. if the output from the scheme comprising the devices D^i is different than the original specification Γ during the n real runs. The game returns zero otherwise.

The game ROB_{Π} contains three phases corresponding to the steps in subsection 2.1.1. First the compilation of the initial specification Γ is done with $\mathsf{TR}(1^k, \Gamma)$, which outputs the specification of the slaves $\{\Gamma_i\}_j$ and the master M. Then the attacker $\mathcal{A}$ receives the security parameter 1^k and the specifications of the slave and the master – the security does not rely on the secrecy of the master specification. $\mathcal{A}$ outputs the physical devices $\{D_i\}_j$ corresponding to the slaves. The

[1] The attacker $\mathcal{A}$ introducing a wireless interface is out of the scope of the solution.

[2] The game has been simplified since it omits the initial vector.

Game $\mathsf{ROB}_{\Pi}(\mathcal{A}, \mathsf{pub}, \Gamma)$:
$(\mathsf{M}, \{\Gamma_i\}_j) \leftarrow \mathsf{TR}(1^k, \ell, \Gamma)$
$\{\mathsf{D}_i\}_j \leftarrow \mathcal{A}(1^k, (\mathsf{M}, \{\Gamma_i\}_j))$
If $\mathsf{T}^{\mathsf{D}_1(.),\ldots,\mathsf{D}_\ell(.)}(1^k, (\mathsf{M}, \{\Gamma_i\}_j)) = \mathsf{false}$ then return 0
$\vec{x}_1 \leftarrow \mathcal{A}(1^k)$
For $i = 1$ to n repeat:
  $\vec{z}_i \leftarrow (\mathsf{M} \Leftrightarrow \mathsf{D}_1, \ldots, \mathsf{D}_\ell)(\vec{x}_i)$
  $\vec{y}_i \leftarrow \Gamma(\vec{x}_i)$
  If $\vec{y}_i \neq \vec{z}_i$ then return 1
  $\vec{x}_{i+1} \leftarrow \mathcal{A}(1^k, \vec{z}_i)$
Return 0.

Figure 2.2: Robustness game from [11]. $\mathcal{A}$ is the adversary, pub are the public parameters of the trojan protection scheme, and Γ is the specification of the circuit to protect. The $\mathsf{View}(x)$ is the set of all input and outputs of the circuit x.

next step is the testing phase, performed by the tester T. It receives the specification $(\mathsf{M}, \{\Gamma_i\}_j)$ and the corresponding devices $\mathsf{D}_i(\cdot)$. If the test does not pass, this means that at least one of the devices did not follow the specification during its t' tests, and the attacker lost the game.

The last step is to perform the n real runs. This is during this phase that the devices $\{D_i\}_j$ should try to deviate from their specifications $\{\Gamma_i\}_j$, resulting in a malicious output $\vec{z}_i$. During that phase, the attacker is selecting the input value of the architecture based on its previous output. If on one input $\vec{x}_i$, the architecture based on M and the slaves $\{D_i\}_j$ returns a $\vec{z}_i$ different than $\Gamma(\vec{x}_i)$, the adversary $\mathcal{A}$ wins the game. Note that due to the majority vote performed by T on the λ slave triplets, at least $\lambda/2$ triplets have to be corrupted.

Hereunder, the probability for an unknown adversary $\mathcal{A}$ to win the game $P(\mathsf{ROB}_{\Pi} = 1)$ is computed. This results in an upper bound for $P(\mathsf{ROB}_{\Pi} = 1)$, and therefore for the probability of the adversary to trigger an Hardware-Trojan that could result in corrupted output.

Security bounds First, the case of a trojan with a constant probability of triggering ϵ is studied: this can be the case of a cheat-code trojan since the input received by the slaves are uniformly random thanks to the MPC protocol. The probability $P(\text{trig}|_{runs} = 1)$ of triggering a trojan during real usage, for a uniform trigger probability ϵ, is given in Eq. 2.1. The trojan must also escape detection during testing. This probability, for t_0 tests, is given in Eq. 2.2. Since $t_0 \leq t$, the bound of the probability of malicious behavior triggering is given in Eq. 2.3 corresponds to the case of the trojan that passes the testing phase and is triggered during the real usage.

$$P(\text{trig}|_{runs} = 1) = 1 - (1 - \epsilon)^n \tag{2.1}$$

$$P(\text{trig}|_{test} = 0) = (1 - \epsilon)^{t_0} \tag{2.2}$$

$$P(\text{trig}|_{test} = 0 \text{ and } \text{trig}|_{runs} = 1) \leq (1 - \epsilon)^t \left(1 - (1 - \epsilon)^n\right) \tag{2.3}$$

An adversary $\mathcal{A}$ should optimize his probability of success by maximizing ϵ in Eq. 2.3. Therefore, he sets ϵ to $\epsilon_{optimal}$ following Eq. 2.4, which results in the probability of success of Eq. 2.5. Since $e^{-1} \leq \left(\frac{n}{t} + 1\right)^{\frac{-t}{n}}$, e being the Euler constant, Eq. 2.5 is bounded by Eq. 2.6. This shows that the testing scheme requires $t \gg n$.

$$\epsilon_{optimal} = 1 - \left(\frac{t}{t+n}\right)^{1/n} \tag{2.4}$$

$$P(\text{trig}|_{test} = 0 \text{ and } \text{trig}|_{runs} = 1) = \left(\frac{n}{t}+1\right)^{\frac{-t}{n}} \cdot \left(\frac{n}{t+n}\right) \tag{2.5}$$

$$\leq \frac{n}{e \cdot (t+n)} \tag{2.6}$$

Finally, as shown in the formal security game in Figure 2.2, an adversary $\mathcal{A}$ wins the game if he can produce the devices $\{\mathsf{D}_i\}_j$ such that they pass t_i tests, and more than $\lambda/2$ of those devices produce an output different from the specification $\{\Gamma_i\}_j$. Due to the majority vote, this probability is then $\left(\frac{n}{e\cdot(t+n)}\right)^{\lambda/2}$. Therefore, the probability that a cheat-code based trojan succeeds in $\mathsf{ROB}_\Pi|_{cheatcode}$ is equal to Eq. 2.7. Note that ϵ is constant due to the input scrambling.

$$P(\mathsf{ROB}_\Pi|_{cheatcode} = 1) \leq \left(\frac{n}{t}\right)^{\lambda/2} \tag{2.7}$$

We can then only consider timebomb trojans to limit the probability of a malicious adversary to lose the game. This happens when the trojan triggers during the real runs n but not during the tests. For one slave circuit, this probability is thus given by $P(v_i \in [t_i+1; t_i+n]) = n/t$, where v_i is the triggering point of the trojan, t_i the number of tests performed, and n the maximum number of runs. Since the full scheme uses a majority vote between λ sub-circuit, the probability of $\mathcal{A}$ to win the game by corrupting more than $\lambda/2$ circuits is $P(v_i \in [t_i+1; t_i+n])^{\frac{\lambda}{2}}$. With this observation and Eq. 2.7, one can find the upper bound of the probability of success of an adversary at the robustness game ROB_Π as in Eq. 2.8.

$$P(\mathsf{ROB}_\Pi = 1) \leq \left(\frac{n}{t}\right)^{\lambda/2} \tag{2.8}$$

Parameters influence Figure 2.3 shows the evolution of security level $P(\mathsf{ROB} = 1)$ for parameters λ and t, and a fixed number of real runs n. One observes that the robustness quickly improves by increasing the number λ of triplets used. This means that in order to reach sufficient security guarantees, increasing the number of test t is useless compared to the gain obtained by adding more hardware. Indeed, in the first case the security level evolves linearly, while in the second case it is exponential.

The implementation in this work is for $\lambda = 1$, i.e. one master with its three associated slaves[1]. Considering a circuit which does not run too often, $n = 1000$ operations is a reasonable goal before having to run more tests. This is what was used to plot Figure 2.3. In this setting, an hour worth of testing of the AES implementation at 10 MHz gives around $t = 10^7$ runs, and thus a correctness up to 10^{-2}, which is clearly not sufficient. With a higher λ and multiple circuits in parallel, the security guarantee rises dramatically: for $\lambda = 10$, 10^7 tests yield a security of 10^{-20}.

Limitations This scheme is asymptotically safe against digital trojan, i.e. cheat codes and time bombs triggers. However, it does not safeguard against physical trojans, for example triggered by temperature change, nor analog ones, like an antenna in each slave that would transmit its share of the secret to a nearby attacker. Another drawback is that the number of runs n is bounded, even if it would be possible to test it again after its n certified runs.

[1] All the extrapolation of the obtained results are straightforward since the λ triplets are independent

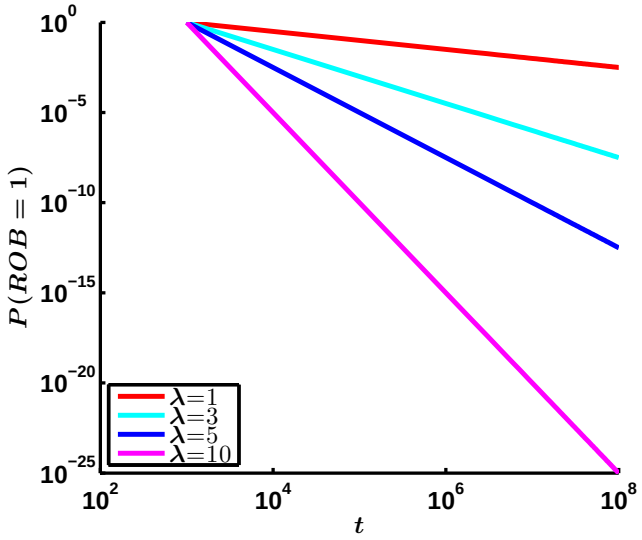


Figure 2.3: Security obtained when running the split circuit 1000 times, as a function of the number of tests t and number λ of sub-circuits.

2.2 Multi-party computation scheme

As mentioned in [11], the performances of the proposed Hardware-Trojan system is highly dependent on the multi-party computation protocol used. Indeed, the bottleneck is the communication speed between the master and the slaves: the key performance metric of the multi-party scheme is the amount of data to be exchanged.

Therefore, this work uses a modified version of the multi-party scheme presented in [2]: it does not require to exchange any element to perform an addition, and only requires to exchange one single field element to perform a multiplication. The trade-off is that this MPC scheme needs to store twice as many bits. However, the block cipher internal data is quite small: for Mysterion it consists of 128 bits for the key and 128 bits for the state, and for AES, $11 \cdot 128$ bits for the pre-expanded key (only $1 \cdot 128$ bits if the key expansion is done on the fly) and 128 bits of state. This makes the storage requirement of the MPC protocol reasonable in the case of the block ciphers, especially since the communication requirements are dramatically decreased.

This section exposes the case of the computation of a function $f(\cdot)$ with input v in the multi-party computation used to build the Hardware-Trojan resilient system. That function $f(\cdot)$ can be a block cipher or any deterministic cryptographic primitive. The way the additions and multiplications are performed in the MPC protocol is highlighted, and the correlated randomness generation is detailed.

2.2.1 Three-way secret sharing over a field

As in [2], the secret sharing is a two-out-of-three sharing. To share an element v, the master computes the share $i \in \{1,2,3\}$ as the tuple (x_i, a_i), with $a_i = v \oplus x_{(i-1)}$ (i wraps around). x_1, x_2 and x_3 are correlated random values, such that $x_1 \oplus x_2 \oplus x_3 = 0$. One share will not reveal the secret v since the x_i are uniformly random. The XOR is then equivalent to a one-time pad [21, p. 32-34]. More practically, in order to share a value v, the master runs Algorithm 3, which requires correlated randomness. The three shares are then obtained by the three slaves.

Algorithm 3: SHARE

Data: v and $x_1 \oplus x_2 \oplus x_3 = 0$
Result: v is shared between the slaves
Master sends $(x_1, v \oplus x_3)$ to Slave_1
Master sends $(x_2, v \oplus x_1)$ to Slave_2
Master sends $(x_3, v \oplus x_2)$ to Slave_3

This sharing is called a two-out-of-three sharing, since any two shares will suffice to reconstruct v, as in Eq. 2.9. This implies that no party should be able to retrieve the share of another one.

$$\begin{aligned} v &= x_1 \oplus a_2 \\ &= x_2 \oplus a_3 \\ &= x_3 \oplus a_1 = x_i \oplus a_{i+1} \end{aligned} \tag{2.9}$$

Once the computation of a function $f(v)$ is performed by the slaves, the master runs Algorithm 4 to reconstruct the output value, where o_i are random correlated values. After receiving the three shares, the computation result $f(v)$ can be reconstructed in three different ways, by using three different pairs of secret-shared values. This allows to perform a simple check that all the reconstruction leads to the same value.

Algorithm 4: RECONSTRUCT

Result: reconstructed value $f(v)$
Master receives $(o_1, f_1(v) \oplus o_3)$ from Slave_1
Master receives $(o_2, f_2(v) \oplus o_1)$ from Slave_2
Master receives $(o_3, f_3(v) \oplus o_2)$ from Slave_3
if $f_1(v) == f_2(v) == f_3(v)$ **then**
  return $f_1(v)$
else
  return *ERROR*

2.2.2 Field addition

Since the shares are obtained from the original value addition, adding two shares together is trivial and consists of an element-wise addition of the tuples[1]. With (x_i, a_i) the shares of v_1 and (y_i, b_i) the shares of v_2, the tuple $(z_i, c_i) = (x_i \oplus y_i, a_i \oplus b_i)$ is the share of $v_3 = v_1 \oplus v_2$. Therefore, for the system to perform an addition, each slave runs Algorithm 5: an addition does not require any communication between the different parties. The field additions are thus free in terms of bits to exchange.

Algorithm 5: ADD

Data: v_1 shared as (x_i, a_i)
  v_2 shared as (y_i, b_i)
Result: $v_3 = v_2 \oplus v_1$ shared as (z_i, c_i)
Slave_i computes $z_i = x_i \oplus y_i$
Slave_i computes $c_i = a_i \oplus b_i$

[1] Note that an addition is equivalent to a $\oplus$ in the bit field

To prove that the above algorithm is correct, note that the sharing of v_3, computed using Eq. 2.9, leads to the expected value $v_1 \oplus v_2$ as shown in Eq. 2.10.

$$\begin{aligned} v_3 &= z_i \oplus c_{i+1} \\ &= (x_i \oplus y_i) \oplus (a_{i+1} \oplus b_{i+1}) \\ &= (x_i \oplus a_{i+1}) \oplus (y_i \oplus b_{i+1}) = v_1 \oplus v_2 \end{aligned} \tag{2.10}$$

To add a constant w to a share, it is sufficient to only add it to the first element of the tuple. The constant addition $v_1 \oplus w$ is performed by computing the tuple $(x_i \oplus w, a_i)$.

2.2.3 Field multiplication

Compared to the addition, the multiplication is a more involved operation[1]. Computing the multiplication $v_3 = v_1 \odot v_2$ requires each slave to send and receive one field element, as illustrated in Figure 2.4.

To compute the shares of (z_i, c_i) representing v_3, the multiplication of (x_i, a_i) and (y_i, b_i) respectively the shares of v_1 and v_2, the slaves compute the three-out-of-three secret as shown in Eq. 2.11. o_i are random correlated values, such that $o_1 \oplus o_2 \oplus o_3 = 0$. To convert that three-out-of-three sharing back to a two-out-of-three sharing, each slave stores the following tuple: $(z_i = c_i \oplus c_{i-1}, c_i)$ and so requires to transfer of field element c_i. So each slave i sends to slave $i+1$ its three-out-of-three share as shown in Figure 2.4. In a more practical way, each slave runs Algorithm 6 to perform a field multiplication.

$$c_i = (x_i \odot y_i) \oplus (a_i \odot b_i) \oplus o_i \tag{2.11}$$

Algorithm 6: MULT

Data: v_1 shared as (x_i, a_i)
v_2 shared as (y_i, b_i)
$o_1 \oplus o_2 \oplus o_3 = 0$
Result: $v_3 = v_2 \odot v_1$ shared as (z_i, c_i)
Slave$_i$ computes $c_i = (x_i \odot y_i) \oplus (a_i \odot b_i) \oplus o_i$
Slave$_i$ sends c_i to Slave$_{i+1}$
Slave$_i$ computes $z_i = c_i \oplus c_{i-1}$

That three-out-of-three secret sharing implies that the three shares are required to reconstruct the multiplied value. In order to show that secret sharing is correct, note the following property:

$$\begin{aligned} a_i \odot b_i &= (x_{i-1} \oplus v_1) \odot (y_{i-1} \oplus v_2) \\ &= (x_{i-1} \odot y_{i-1}) \oplus (x_{i-1} \odot v_2) \oplus (y_{i-1} \odot v_1) \oplus (v_1 \odot v_2) \end{aligned} \tag{2.12}$$

The reconstructed value of the three-out-of-three secret sharing is given by Eq. 2.13. First observe that the correlated randomness o_i is canceled out. Then by using Eq. 2.12, the terms in $(x_i \odot y_i)$ cancels each other out and results in Eq. 2.14. Finally, recall that the terms x_i and y_i are correlated randomness and so their sum is equal to zero. Therefore, the Algorithm 6 is correct to perform a multiplication in that MPC protocol.

[1] Note that a multiplication is equivalent to an AND gate for the GF (2) field, i.e. one bit.

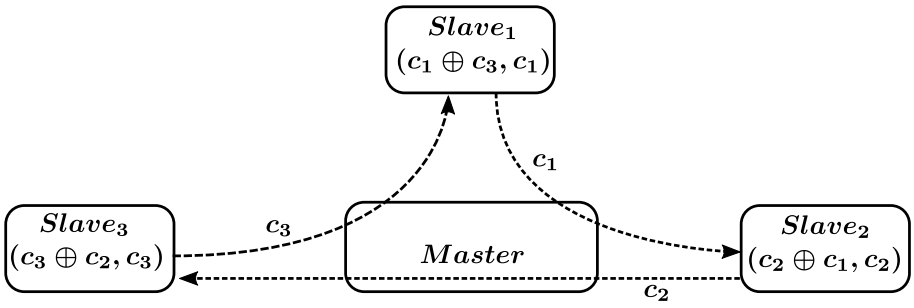


Figure 2.4: Each slave sends one field element to the next, and receives one from the previous.

$$c_1 \oplus c_2 \oplus c_3 = \sum_{i=1}^{3}(x_i \odot y_i) \oplus (a_i \odot b_i) \oplus o_i \tag{2.13}$$

$$\begin{aligned} &= \sum_{i=1}^{3}(x_i \odot y_i) \oplus (a_i \odot b_i) \\ &= \sum_{i=1}^{3}(x_i \odot y_i) \oplus (x_{i-1} \odot y_{i-1}) \oplus (x_{i-1} \odot v_2) \oplus (y_{i-1} \odot v_1) \oplus (v_1 \odot v_2) \\ &= (v_1 \odot (x_1 \oplus x_2 \oplus x_3)) \oplus (v_2 \odot (y_1 \oplus y_2 \oplus y_3)) \oplus (v_1 \odot v_2) \qquad (2.14) \\ &= v_1 \odot v_2 \end{aligned}$$

As in the previous section, there is an easy special case: squaring a secret shared value does not require to transfer any field element. Indeed, to square $v = a \oplus b$ (Eq. 2.15) all the required operations can be applied to a and b independently. Since the multiplication is commutative and since the addition can be seen as a bitwise XOR, one may write Eq. 2.16.

$$(a \oplus b)^2 = (a \oplus b) \odot (a \oplus b) \tag{2.15}$$

$$\begin{aligned} &= a^2 \oplus (a \odot b) \oplus (b \odot a) \oplus b^2 \\ &= a^2 \oplus b^2 \qquad (2.16) \end{aligned}$$

2.2.4 Correlated randomness

One of the requirements of the multi-party computation scheme explained here above is the need in correlated randomness such that $o_1 \oplus o_2 \oplus o_3 = 0$. The solution from [2] is performed in two phases. First a initialization phase where each party generates a random key k_i thanks to a random function and sends it to the party $i + 1$. Each slave i holds two values k_i and k_{i-1}. That algorithm is described in Algorithm 7. Note that the algorithm INIT only needs to be executed once.

The second phase is to generate correlated randomness based on a pseudorandom permutation $F : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ and an $id \in \{0,1\}^k$ with k a security parameter. The output values o_i by the Algorithm 8 are correlated randomness since $o_1 \oplus o_2 \oplus o_3 = 0$. One may note that the slave i can not retrieve o_{i+1} and o_{i-1} since it does not hold all the keys used to generate such a value. In practice, the function $refresh(\cdot)$ can be implemented as a counter, which does not require a transfer. GENERATE is run each time that new correlated random values are required.

Algorithm 7: INIT[2]

Result: Slaves loaded with keys and id
$id \leftarrow \{0,1\}^k$
Slave$_1$ runs $k_1 \leftarrow \{0,1\}^k$; // Key generation
Slave$_2$ runs $k_2 \leftarrow \{0,1\}^k$
Slave$_3$ runs $k_3 \leftarrow \{0,1\}^k$
Slave$_1$ sends k_1 to Slave$_2$; // Key Transfer
Slave$_2$ sends k_2 to Slave$_3$
Slave$_3$ sends k_3 to Slave$_1$
Slave$_1$ holds (k_1, k_3)
Slave$_2$ holds (k_2, k_1)
Slave$_3$ holds (k_3, k_2)

Algorithm 8: GENERATE[2]

Data: k_1, k_2, k_3, id
$F : \{0,1\}^k \times \{0,1\}^k \rightarrow \{0,1\}^k$
$refresh(\cdot)$
Result: $o_1 \oplus o_2 \oplus o_3 = 0$
$id = refresh(id)$; // id refresh
$o_1 = F_{k_1}(id) \oplus F_{k_3}(id)$; // Correlated randomness computation
$o_2 = F_{k_2}(id) \oplus F_{k_1}(id)$
$o_3 = F_{k_3}(id) \oplus F_{k_2}(id)$

However, the proposed algorithm is not suitable to our HT protection scheme. Indeed, a random permutation is used to generate the keys, and should be replaced by a pseudorandom one. Since the tester is PPT, it would not be able to distinguish that pseudorandom permutation from a random one. This means that it could not verify the good functionality of the device. Now suppose that the three slaves did agree on the pseudorandom permutation they would use: each of the slaves is able to get all the k_i and to reconstruct the shared value. This is not acceptable since such an HT could be triggered by a cheat-code.

To solve that issue, it is proposed to replace the INIT function by INIT' described in Algorithm 9. In that way the master selects the keys used by the slaves to generate the random correlated values. The tester will verify that the random correlated values generated by the slaves are correct. The previous issue cannot occur anymore: the slaves are not able to obtain the keys k_i if the slave i is outputting the genuine values. To retrieve the key k_i, the slave i should be triggered by a "time-bomb" or a "cheat-code" on masked value: those two cases are taken into account in the proposed security model [11].

Such a solution implies that the master should generate at least 3 random values (id must not be random). Those three values could be generated by a PUF or by an external randomness source.

2.2.5 Efficient secret sharing

Since the correlated randomness is generated in the slaves, the secret sharing of v performed by the master needs to be highlighted. The master should first run INIT' to generate the random correlated values and then ask the slaves for random correlated values as shown in Algorithm 10. Then the slave i receives masked values $(v \oplus o_{i+1})$ that is randomly distributed and so reveal nothing about the value v. The slave i also keeps the random correlated values o_i he has generated as part of its secret. That methodology is an efficient implementation of the secret sharing

Algorithm 9: INIT'

Result: Slaves loaded with keys and id
Master runs $id \leftarrow \{0,1\}^k$
Master runs $k_1 \leftarrow \{0,1\}^k$; `// Key generation`
Master runs $k_2 \leftarrow \{0,1\}^k$
Master runs $k_3 \leftarrow \{0,1\}^k$
Master sends (k_1, k_3, id) to Slave_1 ; `// Key Transfer`
Master sends (k_2, k_1, id) to Slave_2
Master sends (k_3, k_2, id) to Slave_3
Slave_1 holds (k_1, k_3, id)
Slave_2 holds (k_2, k_1, id)
Slave_3 holds (k_3, k_2, id)

from Eq. 2.9.

Algorithm 10: SHARE' algorithm

1 run **INIT'**
2 run **GENERATE**
3 Slave_1 sends to Master o_1
4 Slave_2 sends to Master o_2
5 Slave_3 sends to Master o_3
6 Master holds (o_1, o_2, o_3)
7 Master sends $(v \oplus o_2)$ to Slave_1
8 Master sends $(v \oplus o_3)$ to Slave_2
9 Master sends $(v \oplus o_1)$ to Slave_3
10 Slave_1 holds $(v \oplus o_2, o_1)$
11 Slave_2 holds $(v \oplus o_3, o_2)$
12 Slave_3 holds $(v \oplus o_1, o_3)$

Its main advantage is that the secret sharing of one *bit* only requires the master to send and receive one *bit*. As shown in subsection 3.3.1, the sharing operation can be performed in one cycle on the master if the bus used is full duplex.

2.2.6 Improvements compared to the original multi-party protocol

The major modification proposed to improve [11] is to modify the MPC protocol used to [2] that was published just after the original proposition. The benefits flowing from that modification are quantified hereunder.

Required throughput The protocol from [2] is more efficient in terms of bits exchanged: only one bit needs to be sent and received to perform one AND operation. In [11, p.8], the computation of an AND gate requires the following exchange with the master to happen two times:

This is a total of $2 \cdot 16$ bits, with $2 \cdot 13$ that have to be sent serially. In the best case, performing a field multiplication requires to send a significant amount of bits while the scheme from [2] requires only one. Therefore, the new multi-party computation protocol introduces benefits on two different points.

First, since the speed bottleneck originates from the communication, an improvement on this front results in direct improvement of the computation throughput. Less data to exchange results

Algorithm 11: Bits exchange required for one AND in the original MPC protocol from [11].

Input: Γ^0, Γ^1 and Γ^2 are the three slave circuits.
Γ^0 sends five bits.
$\begin{cases} \Gamma^2 \text{ receives two bits and then sends one back.} \\ \Gamma^1 \text{ receives three bits.} \end{cases}$
Γ^1 receives one bit and sends two back.
Γ^2 receives two bits.

in a faster multiplication and thus an enhancement on complete block cipher implementation.

Secondly, Algorithm 11 involves a huge amount of communication that will introduce latency in the system: in order to perform a multiplication, a party must wait for the answer of the other ones before ending its computation. On that point, the improvements are also significant since the new multi-party computation is able to perform shared multiplications with zero latency.

Simplicity Another benefit of the protocol is its simplicity: the three slaves implementation is identical, which makes the development straightforward, and the testing phase of the trojan resilient scheme easier. Indeed, since the element transfer are unidirectional[1], those can easily be serialized. There is no need to wait for an answer from another party to send the next field element.

2.3 Generic block ciphers multi-party implementation

The goal of this work is to implement block cipher in the Hardware-Trojan resilient framework exposed above. In order to achieve such a goal, the first point is to clarify and optimize the implementation of those primitives in a multi-party computation protocol. First, the implementation of AES is presented, followed by the lightweight Mysterion block cipher. Then the performances of block ciphers in a MPC protocol are compared. In this section, it is assumed that each slave already has the secret and is able to generate correlated randomness according to the algorithms developed in previous section.

2.3.1 AES for non linear operations reduction

The AES block cipher described in subsection 1.1.2 can be implemented using the multi-party computation scheme from the previous section. The most computing expensive operation is the S-box since it requires an inversion. First the linear operations are exposed followed by two ways to implement the AES S-box .

Linear operations As mentioned in the previous section, the linear operations do not require any data transfer. This means that AddRoundKey, MixColumns and ShiftRows operations are performed as part of a multi-party computation nearly as easily as in a classic AES implementation.

S-box state-of-the-art The only non-linear operation in the AES block cipher is the S-box. As mentioned in subsection 1.1.2, an S-box consists in a field inversion and an affine transformation. Since the costly operation of the S-box is the first step, powerful implementations of $GF(2^8)$ inversion have been proposed in order to minimize the amount of logic gates used in hardware or the table size for software.

[1] always from slave i to slave $i+1$

At the beginning of the 21st century such a solution was proposed to decrease the amount of hardware required to perform the AES S-box. For example, the aim of the proposal in [34, 28] is to optimize the number of logic gates used to perform the field inversion.

At the same time, the side channel community tried to mask the AES S-box[27, 22]. They were interested in decreasing the number of non-linear operations rather than reducing the number of logic gates used. Those two objectives were linked, since by minimizing the number of logic gates, the number of non-linear operations to perform also tends to decrease. The main difference between those two approaches is that the first one is optimizing at the bit level while the second one is optimizing at the field element level.

In the case of multi-party computation scheme, the goal is to minimize the number of bits/-field elements to exchange. By using a very efficient hardware implementation [34], the number of AND gates required is 362, which implies the same number of bits to send. By using an efficient scheme for masking [27], only 5 multiplications in $GF(2^4)$ need to be done. This results in only 5 field elements to send. The focus is set on the masking efficient methods.

This section describes an extension of those works to a multi-party computation. Note that since all the following algorithms are performed in a multi-party protocol, the field multiplication and addition are performed as in Eq. 2.11.

Inversion in $GF(2^8)$ The simplest way to perform an inversion is described in [14]. It uses the fact that the inverse in $GF(2^8)$ can be found by taking the power 2^{254} of that number. Indeed:

$$\begin{aligned} y \odot y^{254} &= y^{1 \mod 255} \odot y^{254 \mod 255} \\ &= y^{255 \mod 255} \\ &= y^0 = 1 \end{aligned}$$

and so we conclude that $y^{254} = y^{-1}$.

Using this identity, an algorithm to compute the 254th power using a minimal number of field multiplications (exponentiations) is used. A way to perform that inversion is represented graphically in Figure 2.5, and using pseudo code in Algorithm 12. The color boxes from Figure 2.5 describe the non-linear operations and therefore, data transfer requirement.

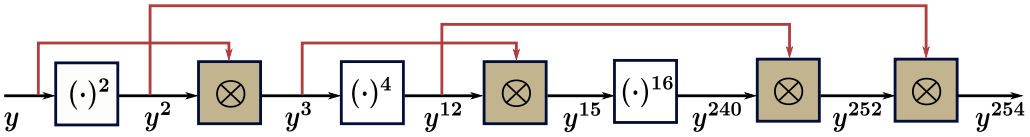


Figure 2.5: Simple inversion scheme over $GF(2^8)$ with non linear colored boxes.

This solution requires 4 field multiplications and 7 squaring in $GF(2^8)$. Notice that in a multi-party computation scheme, the field multiplication requires a data transfer while squaring, which is linear, does not. In the previously described multi-party computation scheme, such an inversion requires 4 field element transfers. Since an element in $GF(2^8)$ is represented by one byte, this solution needs therefore 4 bytes to be exchanged.

Inversion in Composite Field Some improvements to this inversion algorithm can be obtained by changing the representation of the $GF(2^8)$ elements. The main idea is to represent such an element by mapping it to a composite field using an isomorphic transformation $\delta : GF(2^8) \mapsto GF((2^4)^2)$. That transformation allows to represent an element $a \in GF(2^8)$ as $a_1 x + a_0$ with

Algorithm 12: Inversion Algorithm from [14]

Input : y
Output: y^{-1}
$y_2 = y^2$
$y_3 = y \odot y_2$
$y_{12} = y_3^4$
$y_{15} = y_{12} \odot y_3$
$y_{240} = y_{15}^{16}$
$y_{252} = y_{240} \odot y_{12}$
$y_{254} = y_{252} \odot y_2$
return y_{254}

$a_0, a_1 \in GF(2^4)$. Note that the choice of function δ determines the hardware efficiency of the inversion. In this case we choose the function δ from [27]. The mapping $\delta(x)$ can be represented as a matrix operation $T \cdot x$. The matrix is given in Eq. 2.17.

$$T = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{2.17}$$

Using this composite field, the multiplicative inverse in $GF(2^8)$ of $A(x)$ can be computed as

$$A^{-1}(x) = B(x) = b_1 x + b_0 = \frac{a_1 x + (a_1 + a_0)}{a_0(a_1 + a_0) + a_1^2 \omega^{14}} \tag{2.18}$$

with $A(x), B(x) \in GF(2^8)$ and $a_1, a_0, b_1, b_0 \in GF(2^4)$ and ω^{14} a generator of $GF(2^4)$ 0001.

Equation 2.18 is graphically represented in Figure 2.6. In order to invert $x \in GF(2^8)$, the first step is to apply the isomorphic transformation. The second step is to perform some multiplications and additions in $GF(2^4)$. The inversion in $GF(2^4)$ can be implemented in two ways, either using look-up tables or using a similar exponentiation as in Figure 2.5. It would need two multiplications[1]. An important note is that the AES S-box requires an inversion followed by an affine transformation. This affine transformation, denoted α, and the inverse isomorphic mapping, denoted δ^{-1}, can be combined by directly applying the matrix $\alpha \odot \delta^{-1}$.

In the case of multi-party computation, the linear operations (additions, constant multiplication and squaring) are nearly free in an FPGA or ASIC implementation, as they do not imply a data transfer. However, five multiplications in $GF(2^4)$ are needed. Since such an element is represented using four bits, an AES S-box implies twenty bits to transfer. This is a significant improvement compared to the amount of data transfer required for a more naive inversion in $GF(2^8)$ as described in Table 2.1.

[1] in a classical implementation of the AES, using a lookup table for the inversion in $GF(2^4)$ has a negligible footprint since the size of such a table (64 bits) is smaller than the one for $GF(2^8)$ (2048 bits).

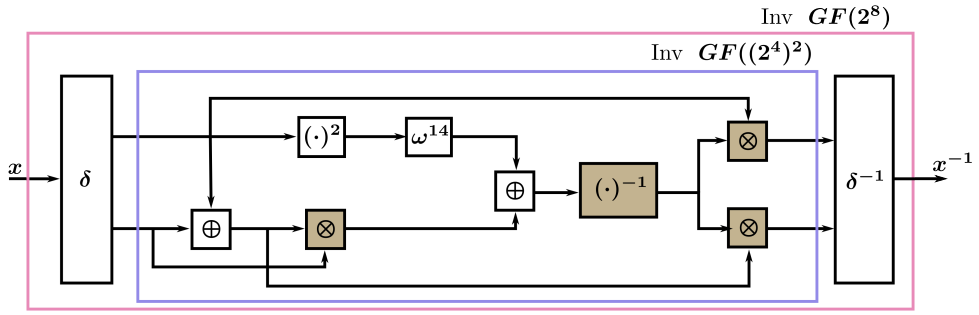


Figure 2.6: $GF(2^8)$ inversion based on composite field $GF(2^4)$, from [27]. All the additions, multiplications and inversion are in $GF(2^4)$. Non linear operations are colored.

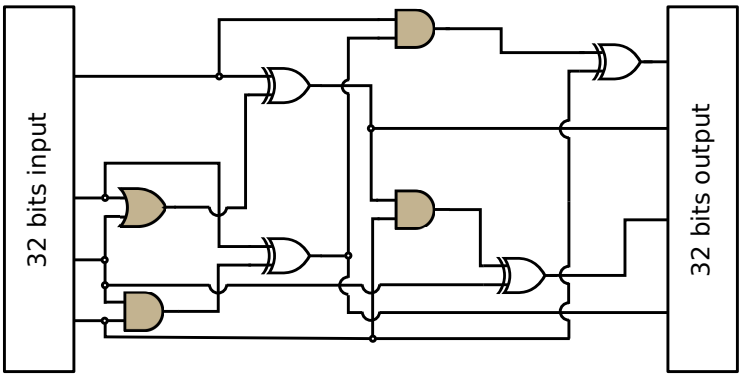


Figure 2.7: Mysterion S-box, as logic gates. Each wire is an 8-bit bus.

2.3.2 Mysterion efficiency for MPC

Since Mysterion is a bitslice cipher, its implementation is considerably simpler and does not require masked operations in Galois Fields. It can be carried out by chaining the bitwise operations of the cipher, and by performing linear multiplications in $GF(2^8)$.

S-box Mysterion S-box is 32 bits wide, and consists of 24 AND, 8 OR and 32 XOR operations. Its implementation is given in Figure 2.7. With the MPC scheme the XOR can be performed on the slave without exchanging any bit, and the OR can be mapped to a AND, as a OR $b = \overline{\overline{a} \text{ AND } \overline{b}}$. Carrying out one S-box requires transmitting 32 bits, and computing the whole round requires the transmission of 128 bits.

L-box The L-box consists of only linear operations, and can thus be computed locally on the shares. The VHDL implementation is straightforward, and consists of plugging the output of one block to the input of the next. That block mainly consists in constant multiplication over $GF(2^4)$ and wire crossing to perform ShiftColumns.

2.3.3 Performances comparison

The main performance characteristics are given in Table 2.1. On one hand, Mysterion is more than two times more efficient than the optimized AES, since the bitslice S-box of Mysterion is performing only one AND per bit. Furthermore, the one time cost of the AES key expansion is not needed. On the other hand, AES is battle tested block cipher and this work presents an

efficient trojan-resilient implementation of it.

The performance metric Bit exchanged per bit encrypted allows to compare the real efficiency of block cipher independently on the number of rounds required. One observe that the intrinsic design of Mysterion allows to reach a better efficiency even if the number of rounds (12) is higher than the AES (10).

Cipher	**Rounds**	**Bits exchanged per round**	**Bit exchanged per bit encrypted**
AES with $GF\left(2^8\right)$ **inversion**	10	512	40
AES with $GF\left((2^4)^2\right)$ **inversion**	10	320	25
Mysterion	12	128	12

Table 2.1: Cipher comparison in MPC

3

Efficient Hardware Architecture for Block Cipher Implementation

The mathematical results from previous sections highlight the possibility of implementing block cipher using multi-party computation efficiently. This section proposes a way to implement the block ciphers AES and Mysterion using the MPC scheme from section 2.2, in a Hardware-Trojan resilient way. Since Mysterion can be more efficient than AES, its implementation benefited from additional hardware optimization compared to AES with $GF\left(2^8\right)$ and $GF\left((2^4)^2\right)$. This choice has been confirmed with a first rough hardware implementation.

This section first explains the way the different sub-circuits are interconnected, then the layout of the master and slave circuits. Finally, the simulations results are laid out and the performances of the block cipher are compared in terms of required cycles.

3.1 Phases of encryption

As described in chapter 2 an encryption is composed of three distinct phases:

1. *Loading*: is the first phases that implies to share in plaintext according to Algorithm 3. In that phase the master is sending masked plaintext to the slaves while those are sending correlated randomness. This operation is only depending on the plaintext size n: this requires that the master receives and sends n bits. Those operations can be done concurrently in case of a full duplex bus, as shown in subsection 3.3.1.

2. *Rounds*: implements the block ciphers rounds using multi-party operation as addition (Algorithm 5) and multiplication (Algorithm 6). Of course, depending on the block cipher, different operations need to be performed. AES is described in section 2.3 while the ones of Mysterion are in subsection 2.3.2. During that phase, the slaves are connected through the master that only performs forwarding. It also performs some synchronization between the different slaves through SCK, nSS and IRQs.

3. *Reconstruction*: implements the reconstruction presented in Algorithm 4. The slaves are sending their own secret tuple (x_i, a_i) and the master outputs the reconstructed cipher text and an error bit[1]. Such an operation requires the slaves to send $2n$ bits: however the master can perform the reconstruction in an efficient way as shown in Figure 3.5.

The proportion for those different operations are shown in Figure 3.7: the impact of loading and reconstruction is negligible in the case of AES $GF\left(2^8\right)$ and while is it not in the case of Mysterion. Indeed around 20% of the cycles are spent on those loading and reconstruction phases.

[1] high if the secret were correctly shared, low if not.

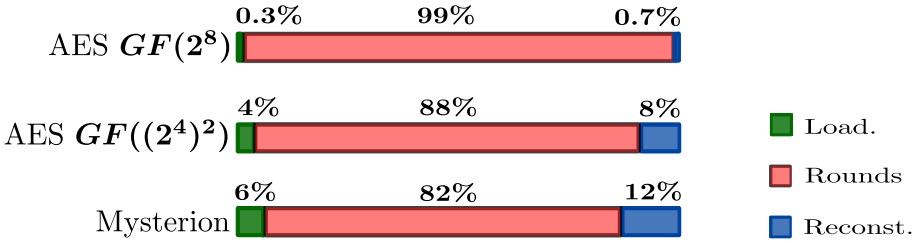


Figure 3.1: Percentage of cycle used between the three difference phases of an encryption for multiple block cipher implementation.

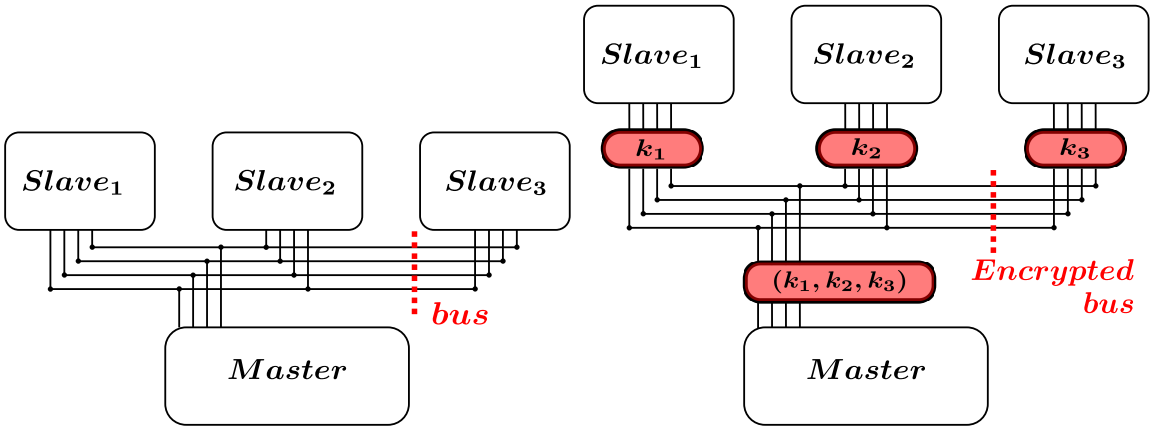
3.2 Bus architecture

One important element in building a digital system is the way the elements are interconnected. This choice influences the security guarantees, the complexity of the elements and amount of I/O required on the chips. This section highlights some issues on the interconnection and the choices that have been made in order to achieve the required security level and good performance.

3.2.1 Constraints on connections layout

In a classical digital system, the elements are connected in a way described in Figure 3.2a (i.e using PCI or SPI bus). In that case, each slave has a dedicated address. If the master wants to send data to a slave, it broadcasts data on the bus. The slaves are listening for messages, and treat them only if they are referring to their address.

However, in that case, all the slaves can read all the messages on the bus (sniffing). If a slave has such a power, it is able to recover the plaintext. This could lead to a cheat-code trigger and is thus not allowed.



(a) Clear bus. **(b)** Encrypted bus.

Figure 3.2: Interconnection between master and slaves with clear or encrypted bus. The yellow boxes are dedicated to encryption.

A solution to such a problem would be to encrypt the data sent on the bus, as proposed in Figure 3.2b. The master would encrypt the data with a different key for each slave. The first issue is to select the key used to encrypt the bus. A simple solution would be to send in clear a key k_i to a slave using the bus. However all the other slaves are able to sniff the key exchange and read the plaintext. But since the first step of the security scheme is to test each slave independently,

the bus encryption key could be pre-loaded during the testing phase, when the other slaves are not connected to the bus.

However this solution would require to perform a trusted encryption in the dedicated module (yellow boxes in Figure 3.2b). Since this is the goal of the entire system, the bus encryption solution is not applicable in this work.

3.2.2 Choice for parties interconnection

Another solution is to connect the master to each slave separately, as shown in Figure 3.3. The proposed system uses an independent SPI interface between the master and each slave. In this case the slaves are not able to sniff data sent to other slaves. However the drawback of such a layout is the increased amount of I/O resources required on the master. Indeed, the number of required I/O pins is increasing linearly with the number of slaves.

The choice of SPI is motivated by two different points. First, SPI is a lightweight protocol. This property is critical since the master needs to be as small as possible in order to be able to verify its functionality easily. The second point is that the SPI is a full-duplex communication. As shown in the following sections, this allows to avoid the master to store data sent by one slave[1].

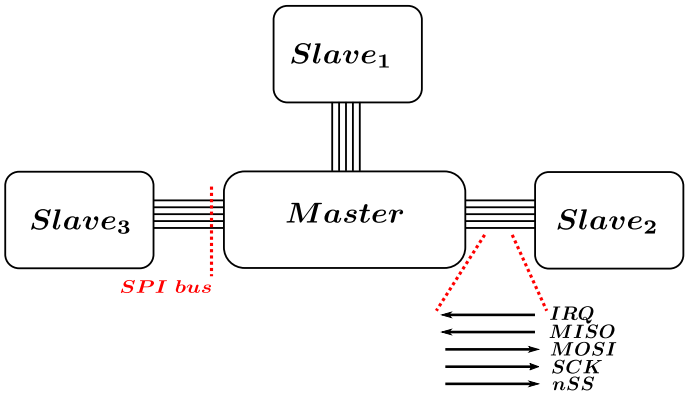


Figure 3.3: Hardware-Trojan resilient connections between master and slaves.

In addition to the wires dedicated to data transfer, an additional wire is used to raise interrupts from each slave to the master. This interrupt request line (IRQ) is mainly used to synchronize the different parties. For example, it can be used to signify to the master that a slave is ready to send data.

3.3 Master hardware description

The master is the critical part of the system, and needs to be minimized in size. Its tasks, described in section 3.1, are to load initial secret into the slaves, forward the message between the slaves and finally to reconstruct the processed secret.

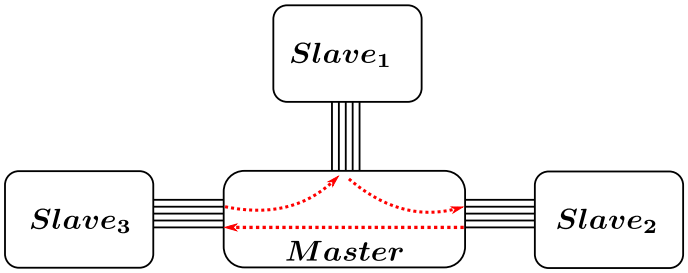
This section proposes to first study the hardware required at the master side to perform those phases. Then the amount of logic gates are described inside the master in order to verify the hypothesis of its size.

[1] In addition, the SPI protocol does not require special I/O pins, which allows the system to be more general.

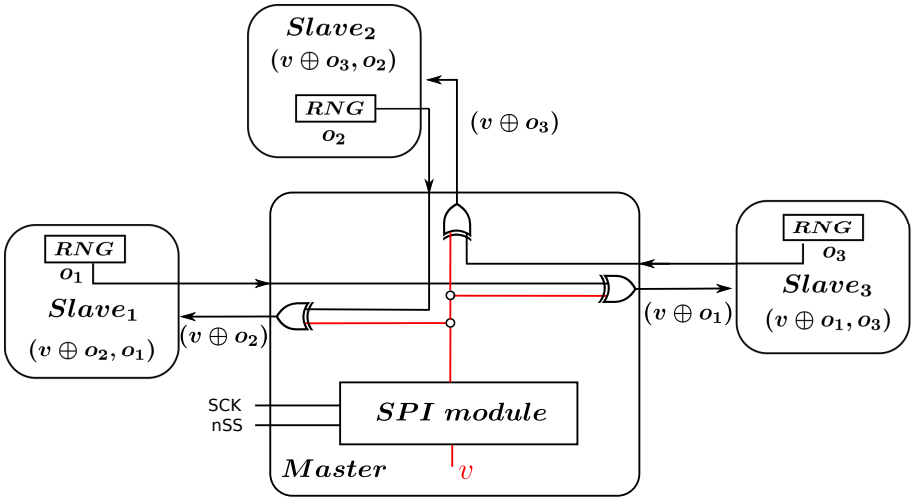
3.3.1 Hardware involved in loading

Here, the case of sharing a bit v between the slaves is investigated. That value v can also be a word of fixed length of a field element without loss of generality. That v can be a key for encryption or a plaintext. As a reminder from subsection 2.2.4, the slave i generates the correlated random value o_i and keeps it as part of its share. Then, o_i is sent to the slave $i+1$ in order to compute $v \oplus o_i$.

The connections of the blocks involved in the loading phase are illustrated in Figure 3.4b. The master only contains one SPI module that generates the control signals *nSS* and *SCK*. That module is operating in a standard way[1]. The output of that module, *MOSI*, is directly masked with the random correlated value o_i sent by a slave. This masking is purely combinatorial logic, and does not require any registers.



(a) Slaves connections to perform a field multiplication. The master is only forwarding the data without processing.



(b) Interconnection to load a bit v from the master to the slaves. The RNG blocks are used to generate correlated randomness inside the slaves, as described in subsection 2.2.4.

Figure 3.4: The two modes of connections for the field multiplication and the loading operations.

This solution has multiple advantages. First, the amount of hardware required is extremely low. Indeed, in addition to the SPI module, only 3 XOR gates are needed. Secondly, there is no cycle delay between the first slave sending the bit v and the second slave receiving the masked

[1] Description can be found at `https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus`. The proposed solution is a slight modified version of SPI as shown later

value.

However, the main drawback is that the propagation delay between one slave to the other is doubled compared to a communication between the slave and the master. This may result in a decreased SPI clock frequency. However one could add 3 registers in the master in order to pipeline the o_i sent by the slaves. This solution would introduce a one cycle delay, and potentially double the bus operating frequency.

3.3.2 Interconnection for the forwarding phase

For the forwarding phase, the master is just connecting the different parts of the slaves. A synchronization is required between the different parts of the system. That synchronization is obtained with two signals: IRQ indicates to the master that the slave is ready to perform the next operation, and nSS indicates to slave that the master has answered its request.

The communication protocol used in this implementation is not exactly an SPI, since it does not shift in an address before shifting in or out the data. In this case, the data is identified by the order in which it is sent, since the three slave work synchronously. This also allows to the SPI to send variable length words since each party of communication what word will be send next. To forward data, the master only needs to generate control signals of the SPI communication, namely nSS and SCK.

3.3.3 Efficient hardware secret reconstruction

As well as for the loading phase, the secret reconstruction is quite tricky. Both of those phases require computation inside the master, which needs to output the reconstructed value v as well as a flag indicating is that value can be trusted. A contrario to the loading phase, the master does not exploit anymore the fact that the bus is full duplex. In this case there is no requirement for the bus. In this section, different architectures are proposed to reconstruct a value v shared as $v = a_i \oplus x_{i+1}$ (Eq. 2.9) according to the bus architecture. For this section we denote x_i^j as the jth value hold by the slave i.

A first, but simple solution would be that the master requests all the (x_i^j, a_i^j). Once it holds all those values, it can reconstruct all the v^j by operating three 128-bit xor and compare the outputs. Such a simple solution is quite hardware consuming since it requires 384 XOR gates, 768 registers and a comparator of 128 bits. Some other solutions can achieve the same result using a reduced amount of hardware.

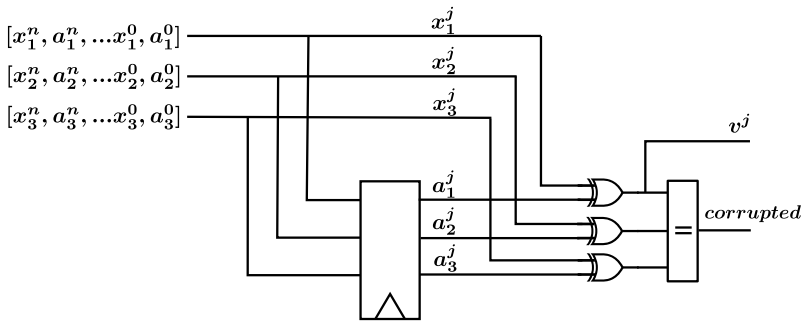


Figure 3.5: Efficient reconstruction of v in master in the case of a serial bus.

In the case of a serial bus, the master receives from the slave bit sequences denoted $[x_i^n, a_i^n, \ldots x_i^1, a_i^1]$. Note that by taking the two first bits sent by the slaves, it is possible to reconstruct one bit of v. This principle is illustrated in Figure 3.5. In such a architecture, the bit sequence is delayed in order to have at the same time x_i^j and a_i^j available in the circuit. This allows to directly reconstruct a bit v^j and perform the comparison. This solution implies to use the 3 registers, 3 xor and one comparator of 128 bits, which is a significant improvement compared to the simple solution.

In the case of parallel buses, some improvements can be made. Indeed, the slaves are able to send values on 2 wires at the same time. One wire is sending the sequences $[x_i^n \ldots x_i^1]$and the other $[a_i^n \ldots a_i^1]$. Since the values x_i^j and a_o^j are directly available in the circuit, there is no more need to delay the signals. This solutions is shown in Figure 3.6. The improvement from Figure 3.5 is only to remove the register used. Of course there is a lot of intermediate trade-off. For example, by increasing the bus width, more XOR gates and comparators are required but the time to reconstruct the secret decreases.

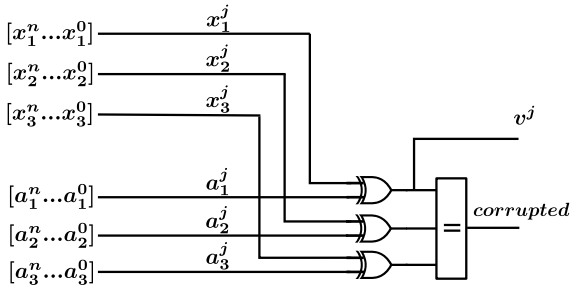


Figure 3.6: Efficient reconstruction of v in master in the case of a serial bus.

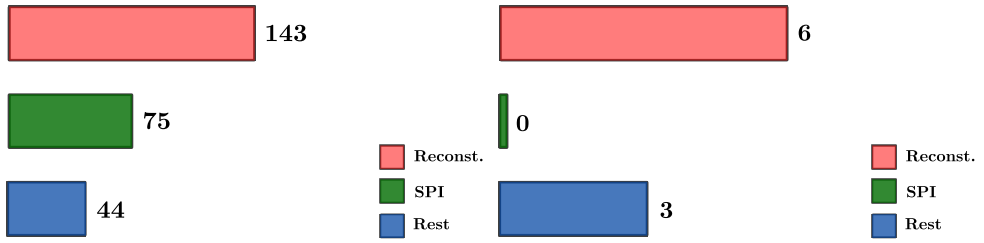
3.3.4 Trusted gates reduction

As mentioned above, the amount of hardware inside the master is critical. This section proposes to observe the logic consumption of the master. Those results are obtained thanks to the synthesis of the master using the tool Design Vision from Synopsys, and by parsing the obtained netlist[1].

Figure 3.7 analyzes the proportion of XOR gates and registers. First observe on Figure 3.7a that the total amount of registers used in the master is 262. In those registers, 143 are used in the reconstruction modules. Among those, 128 are used to store the cipher text and could be deleted by streaming the secret out of the master. Three others are used to delay the received bits as shown in Figure 3.5. The remaining registers are dedicated to control purposes. If the 128 registers used to save the cipher text are removed, we observe that the contribution of the SPI module to the registers consumption is the most significant compared to the rest of the system. Those registers can not be removed since at least one communication module is needed.

Figure 3.7a compare the amount of XOR gates used to in the system. One may observe that 6 XOR are used in the reconstruction of Figure 3.5. Indeed three are used for the actual XOR and the three others are dedicated to the comparison. The three register in the rest of the systems are

[1] A tutorial can be found at: `http://beethoven.ee.ncku.edu.tw/testlab/course/VLSIdesign_course/course_96/Tool/Design_Vision_User_Guide.pdf`



(a) Register used (total of 262). **(b)** XOR gates used (total of 9).

Figure 3.7: Number of logical element used in the master. Blue corresponds to logic dedicated to the reconstruction, Red to the SPI module and Orange to the rest of the system.

the one used at the loading phase (Figure 3.4b).

That result is a key point in the Hardware-Trojan framework: it is possible to implement Hardware-Trojan resilient system by significantly decreasing the amount of trusted logic gates compared to a standard block cipher hardware implementation [1]. As a comparison, lets take a standard AES Sbox: [34] is building its S-box and inverse with around 600 logic gates while our construction is using only around 300 by taking into account all gates.

In addition, a complete AES does not contains only one S-box but generally multiple in parallel as well as other operations as MixColumns which requires around 140 gates [26]. A classical AES with 16 S-box in parallel and one MixColumns would include around 9740 gates which is 33 times larger than our trusted master even when omitting the KeyAddition, ExpandRoundKey blocks.

Another hypothesis is that the master complexity is independent of the slaves complexity. In practice, this is the case except for a counter of the transfers that the master has to perform. For example, for Mysterion and AES, the master has to forward two times 64 bits per rounds. However, AES has 10 rounds while Mysterion has 12. One may conclude that this hypothesis is therefore true up to a counter that grows in $\mathcal{O}(\log(n))$ with n being the number of transfers to perform.

3.4 Slave description

Two versions of the slave are presented in this work, one for the AES encryption and one for Mysterion. This section presents the design choices and high level characteristics of the implementation.

The high-level slave layout is given in Figure 3.8. It follows closely the description of the AES blocks of subsection 1.1.2. Only the data loader and the S-box, the non-linear part of the cipher, are connected to exchange data with the other slaves, via the SPI bus. A finite state machine controls the sequential activation of the different blocks, and hands over the control of the SPI to the currently activated function.

The slave layout is given in Figure 3.9. At a high level, it is very similar to AES, and only the loader and S-box need access to the I/O.

[1] only the case of AES is discussed due to the lake of hardware Mysterion implementation.

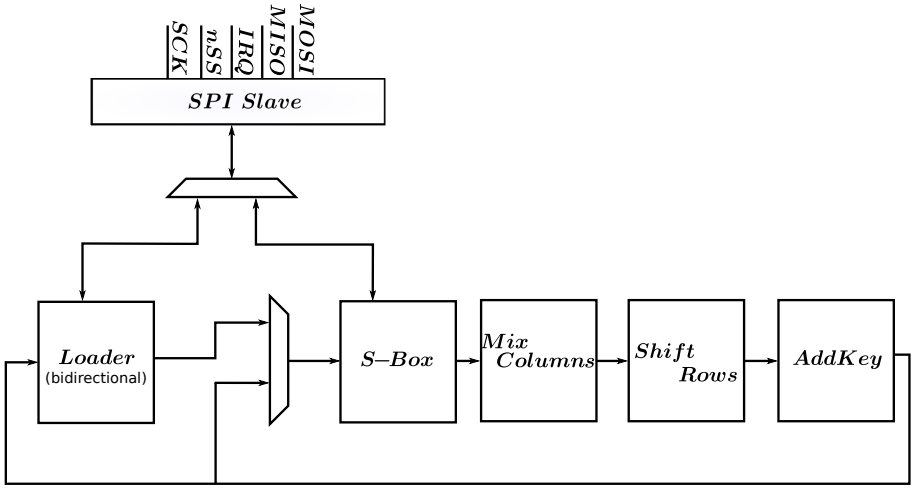


Figure 3.8: AES slave layout.

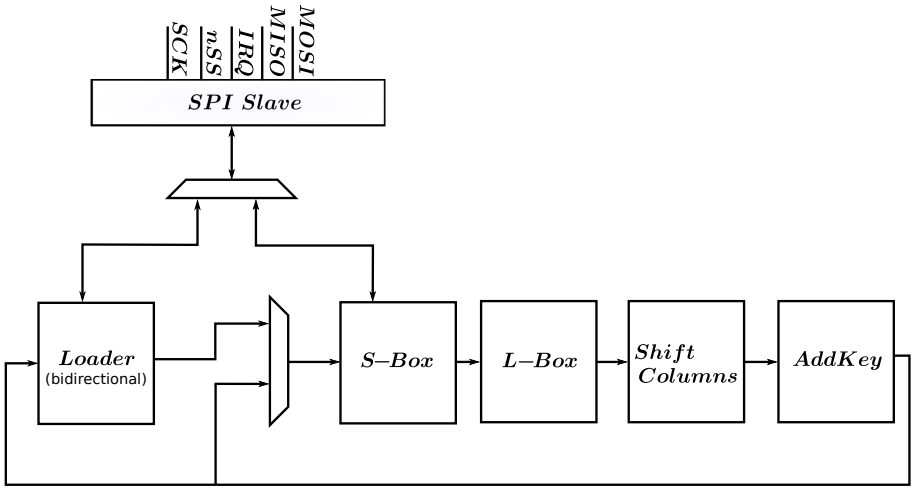


Figure 3.9: Mysterion slave layout.

3.5 Performances

When building a system, the performance analysis is one of the most important point: it allows to validate the choices that have been made. This section first describes the performances on the actual system and then extrapolate those results in the case of an ideal bus.

3.5.1 Performances modeling

Eq. 3.1 is proposed to model the system performances: the functions C represents the number of cycles to run an implementation θ of algorithm α, β is a parameter representing the bus performances, for example in [bit/cycle] with cycle being internal of the chip(Table 3.1). That parameter β is the bus quality since it represents the maximum amount of bits that can be exchanged during an internal clock cycle. For example, if $\beta = 0.25$, one bit transfer takes 4 cycles and a complete 128 bits state transfers consumes 512 cycles. Therefore, an high β hold for a well performing bus while a low β represent a low quality communication channel.

Function T stands for the number of cycles used to transfer the amount of data required by the algorithm α on a bus characterized by β. It is important to note that such a function does not depend on the implementation and is so a reference lower bound in term of cycles for a given bus.

$$C(\alpha, \theta, \beta) = T(\alpha, \beta) + L(\alpha, \theta) + Con(\theta) \tag{3.1}$$

Functions L and Con respectively correspond to the amount of cycles spent in linear operations and control. Linear operations are required by the algorithm α, but the implementation θ can influence the cycle required to perform them. A simple example is parallelizing operations: by increasing the amount of hardware, an implementation θ_1 would be faster than θ_2 which would reduce the amount of logic gates used to perform the same operation.

The time spent in control operations mainly depends on the quality of the implementation: a good one would reduce the amount of control cycles. Note that those two functions do not depend on the bus quality β since the multi-party protocol allows not to exchange any bit for linear or control operations.

Parameter	Description
C	Total number of cycles
T	Number of cycles dedicated to data transfer
L	Number of cycles dedicated to linear operations
Con	Number of cycles dedicated to control purpose
$\boldsymbol{\alpha}$	Algorithm implemented in the MPC protocol
$\boldsymbol{\theta}$	Number of cycles dedicated to control purpose
$\boldsymbol{\beta}$	Number of bits exchanged on the bus during one internal cycle

Table 3.1: Parameters and variables description for performances analysis.

3.5.2 System performances

As mentioned earlier in Eq. 3.1, the performance depends on three different parts: T for transfer, L for linear and Con for control. This section proposes to observe the interaction and the choices that have been made based on the comparison in Figure 3.10. In this figure, the bus is a single wire SPI with variable frequency: results and extrapolation to larger bus (larger β) are shown later.

Transfer has a huge impact on the system performance: the metric β, defined as the bits exchanged per cycle, is proposed to analyze its effect. This criterion represents how fast the communication is compared to the internal frequency. In practice, this ratio is defined by the communication medium characteristics and critical paths in the system.

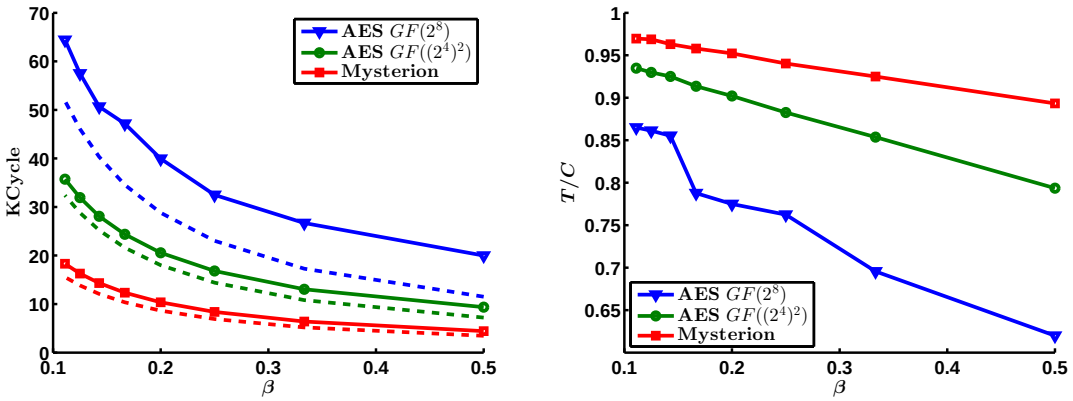
The lower bounds (dashed lines) in Figure 3.10a correspond to the function T for different block ciphers and therefore to the lower bounds in terms of cycles for an entire encryption. As expected from Table 2.1, the AES implementation using an inversion in $GF\left((2^4)^2\right)$ achieves better results compared to the inversion in $GF\left(2^8\right)$, and Mysterion is the most efficient block cipher. The difference between the throughput of those implementations mostly relies on the improvement of T.

Linear includes all the linear operations required to compute the block cipher. Those contain the squaring, constant addition/multiplication and wire crossing. The choice has been made to use parallelism to decrease the L function.

Control includes all the cycles not involved in useful computation and needs to be as low as possible: the control function Con determines how good an implementation is in terms of cycle efficiency.

The difference between the solid and dashed lines in Figure 3.10a is due to L and Con. Observe that AES using inversion $GF\left((2^4)^2\right)$ and Mysterion are using a bus with a variable word length while AES based on $GF\left(2^8\right)$ does not. In the first case, the communication can transfer any word length (i.e. 32, 64 or 128 bits) which may be useful to transfer an entire state, while in the second case, the transfer is only done one byte at a time. In that case, the bus needs an increasing amount of control, which explains the larger difference between the performance and the lower bound for AES in $GF\left(2^8\right)$.

In addition, one may observe that the difference between the lower bound (dashed lines) and simulation results (solid lines) is very small for Mysterion. Indeed it has benefit from additional improvement at the control and linear level. In the case of Mysterion, the linear and control operations take 471 cycles with 261 involved in linear and 210 in control (Table 3.2). Those two could certainly be optimized in future work to lead to a maximum of 44% improvements in the case of a larger bus width as shown in Table 3.3.



(a) Cycles per encryption.

(b) Bus Occupation.

Figure 3.10: Performances comparison between the three proposed cipher: (a) in terms of cycles (b) in terms of bus activity for bus quality $\beta[bit/cycle]$. Dashed lines represent low bound for performances of a block cipher at a given bus quality.

Figure 3.10b illustrates the saturation effect of the bus when its quality β is degraded: the graph is showing the ratio $T(\alpha,\beta)/C(\alpha,\beta,\theta)$ which becomes closer to one for a smaller β. The saturation is due to the increasing value of T term compared to L and Con which are constant with the bus quality β. This highlights the intuition that the bus performance is the bottleneck of the entire system. Due to this saturation effect, one may conclude that pipelining encryptions would not be useful since the communication bus is already used for more than the half of the time.

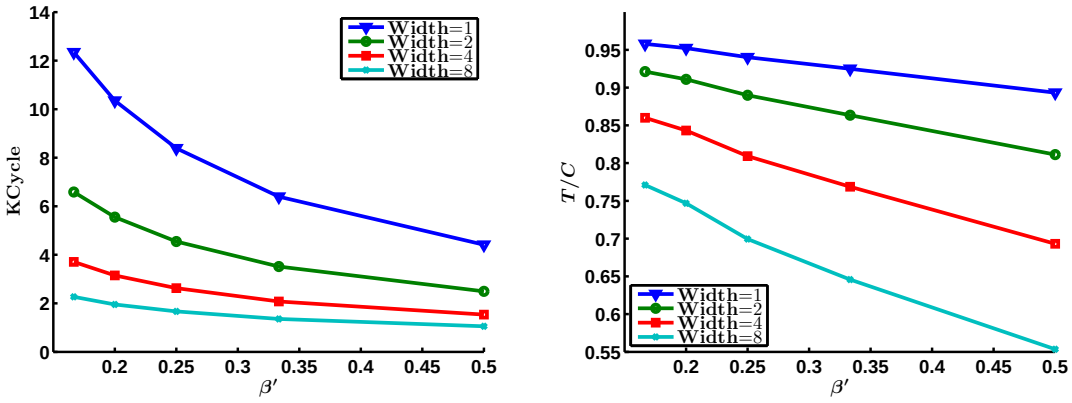
Block Cipher	C [cycle]	T [cycle]	T/C [%]	L + Con [cycle]
AES $GF\left(2^8\right)$	19978	12384	62	7594
AES $GF\left((2^4)^2\right)$	9356	7424	79	1932
Mysterion	4415	3944	90	471

Table 3.2: Block cipher performances at $\beta = 0.5$, graphically represented in Figure 3.10.

3.5.3 Bus width influence

In order to get better performance, it is proposed to use a SPI communication with an increased number of parallel wires. The width of the bus corresponds to the amount of wires that are linking two parties in one direction. Since the communication is using a SPI bus, a width n corresponds to a n-bit MISO and n-bit MOSI, for a total of $2n$ wires dedicated to data transfer.

Figure 3.11 compares the performances of Mysterion for different bus width in function of β', the bits transferred per internal cycle and per wire. The simple conversion $\beta' = \beta/(width)$ is written. expected, Figure 3.11a shows that by increasing the width, the throughput is increased linearly. The best performance is obtained for $\beta' = 0.5$ and the widest bus, with $n = 8$. In that case, a single encryption takes 1055 cycles with around 450 dedicated to L linear and Con control terms.



(a) Cycles per encryption.

(b) Bus Occupation.

Figure 3.11: Performances comparison for different bus width on Mysterion: (a) in terms of cycles (b) in terms of bus activity for bus quality $\beta'[bit/cycle]$ of a single wire. The exact values are given in Table 3.3.

Since the terms L and Con are independent of the bus width, and since the term T is decreasing with the width (β increased in Eq. 3.2), the bus is less saturated in case of a wider interface, as shown in Figure 3.11b. In the best condition, i.e. at the highest throughput, the bus is busy around 55% of the time during an encryption. Therefore, it should be possible to run multiple encryptions at the same time: this would, in the best case, introduce a gain in performances of around 80%, with only a small increased amount of logic gates and storage, since it uses idle resources. Those optimizations are left for future work.

Bus	β [bit/cycle]	**C** [cycle]	**T** [cycle]	**T/C** [%]
Width = 1	0.5	4415	3944	90
Width = 2	1	2495	2024	81
Width = 4	2	1535	1064	69
Width = 8	4	1055	584	55

Table 3.3: Bus width influence on Mysterion performances at absolute $\beta = 0.5$ [bit/cycle].

3.5.4 Results extrapolation

Since SPI is not a very efficient bus, this section proposes to extrapolate the model from Eq. 3.1 for faster transfers. The extrapolation is based on the lower bound function T that is computed in Eq. 3.2 with β the bits exchanged per cycle and $n(\alpha)$ the number of bits to exchange for the algorithm α.

$$T(\alpha, \beta) = \frac{n(\alpha)}{\beta} \tag{3.2}$$

By increasing β, the lower bound is decreasing in β^{-1}. As shown in Figure 3.12, with a bus exchanging 32 bits per cycle (i.e PCI), the expected performance is under 100 cycles for Mysterion. However, with such an efficient bus, the impact of L and Con from Eq. 3.1 and dominate the system performances.

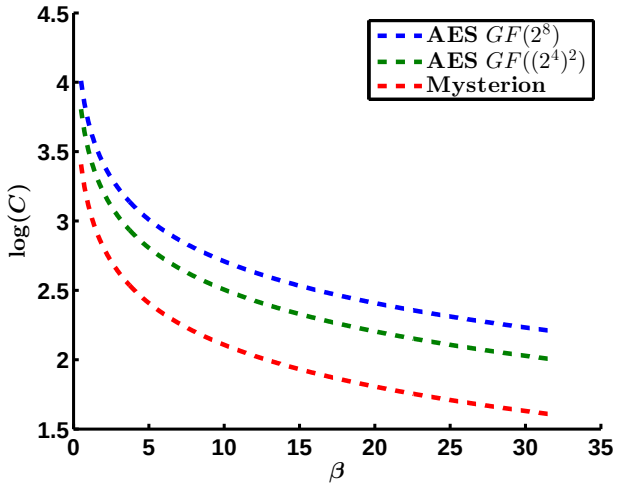


Figure 3.12: Extrapolation for performances according to bus efficiency β.

3.5.5 Randomness generation performance

Note that by increasing β as in the previous section, the correlated pseudo random generator F'_k described in subsection 2.2.4 must be performing fast enough as not to be the system bottleneck. Intuitively, the PRG should generate as much bits as the bus can send. Formally, this is written in Eq. 3.3, with $R(F'_k)$ the throughput of the F'_k, i.e. the number of bits output by the PRG, divided by its latency in cycle. N is the number of PRG bits required for an encryption and $C(\beta)$ is the number of cycles taken by an encryption. The right part of the equation is representing the performances obtained with an ideal PRG. In such a case, the PRG is not the bottleneck of the system.

$$R(F_k') \geq \frac{N}{C(\beta)}\bigg|_{R(F_k')\to\infty} \tag{3.3}$$

In practice, the lower bound can be found by replacing the PRG with a module that returns deterministic bits in one cycle. Applying this to the case of Mysterion, with a $\beta = 4$ (bus width = 8), the lower bound is of 1.6 pseudorandom bit per cycle. Since the PRG used is characterized by $R(F_k') = 2.3$, one deduces that it is correctly sized since it is close to the lower bound defined in Eq. 3.3.

Sizing correctly the PRG is important: on one hand, if it is too fast, it would waste area that could be better used to speed up the linear phases. On the other hand, if it is too slow, it then becomes the bottleneck of the entire system. The PRG used is implemented as two AES in counter mode. The implementation is outputting 128 bits in 55 cycles and takes already around 30% of the logic required inside the slaves, as discussed in section 4.3. This leads to the conclusion that with an increased bus quality β, the PRG should be sized accordingly and would therefore require an increased amount of hardware resources.

3.5.6 Internal throughput influence

The previous results are only comparing the performances in terms of internal cycles, but the real performance of the system is defined in time. This section discusses the influence of the internal throughput on the performances of the system at a fixed bus quality. This allows in example to discuss the gain obtained by increasing the speed of linear operations on a physical implementation as exposed in following chapter. Therefore, two frequencies are defined: f_{bus} being the bus frequency and f_{int} being the internal frequency of different parties.

Figure 3.13 shows the influence of the internal frequency f_{int} for fixed bus performance: changing the internal frequency allows to directly influence the internal throughput. The internal throughput can also be increased by introducing parallelism in linear operations and not only by increasing the frequency. Figure 3.13 takes as reference the best case in terms of internal cycles from Table 3.3. This figure shows the impact of internal logic throughput on the complete system performances.

On that figure, the term T_t, which represents the time spend in transfer operation remains constant even if the internal throughput increases since it is limited by the bus performances. However, the impact of $L_t + Con_t$ on the final throughput is decreased if the internal frequency is increased, and therefore its throughput is enhanced.

In the case of Mysterion, the linear and control operations take 471 cycles with 261 involved in linear and 210 in control (Table 3.2). Those two could certainly be optimized in future work to lead to a maximum of 44% improvements, as shown in Figure 3.13. Such improvements would significantly reduce the difference between the theoretical performances lower bound and the electronically implementation.

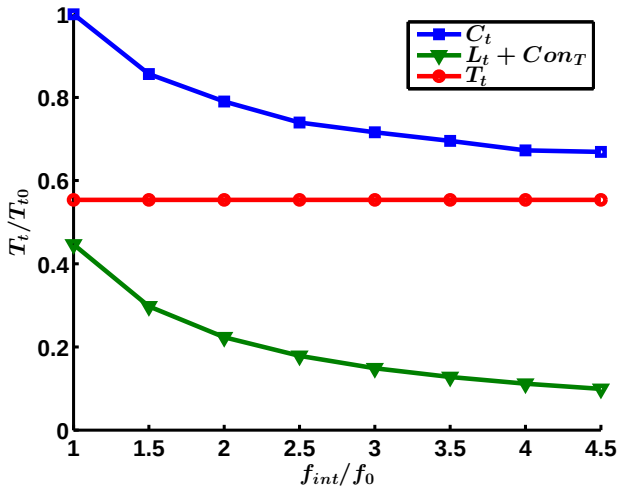


Figure 3.13: Performance in term of total time for fixed bus frequency $f_{\text{bus}} = f_0/2$ as a function of the internal frequency f_{int}.

4

Demonstration Board for Hardware Trojan Resilient Cryptography

The multi-party scheme described in subsection 1.1.5 is implemented for Mysterion on a PCB with four FPGAs, as a proof of concept. The results given in this section are for Mysterion unless mentioned otherwise. The extrapolation for AES is straightforward thanks to the simulation results and is explicitly mentioned. First the experimental board is exposed, followed by the performances in term of throughput against variable parameters then a discussion about the hardware resources usage is proposed. This chapter ends with an application note and extrapolation results for future implementations.

4.1 Chip description

The board design includes four FPGAs with three of them connected to the last one to implement the proposed Hardware-Trojan resilient scheme. The Xilinx Spartan 6 LX9 FPGAs were chosen since they come in a convenient TQFP package that is easy to solder in a PCB. The Spartan 6 LX9 FPGA [45] has the advantage of being large enough to contain the proposed design and of a reduced price: around €15[1]. The main information of this section are summed up in Table 4.1 and the resulting demonstration board in Figure 4.1.

Board Design The board as been design thanks to the PCB and schematic CAD tool *Eagle* following the Xilinx recommendation [46]. First the power network management manual [48] describes the decoupling network that was used on the selected FPGAs. Those are relatively small compared to other FPGA of Spartan 6 family. The FPGAs have different supply voltages: one for the internal logic ($1.2V$) and one the I/O ($3.3V$) for a total board power consumption of around $0.4W$. The power networks need to be powered by an external generator.

The I/O manual [47] has been used to design the connection between the FPGAs. The I/O used are either LVDS (Low Voltage Differential Signaling) which is a differential pair I/O standard or LVCMOS(33) (Low Voltage Complementary Metal Oxide Semiconductor) which is a single-ended. Those two have the main advantage of being build in Spartan 6 FPGA and therefore do not require additional hardware. The master is connected using LVDS standard to slave 1 and 2, and with classical single-ended pins for slave 3, since the LVDS-capable I/O banks of the master FPGA were exhausted. Splitting the LVDS capable banks across the three slaves would make the routing more complex. All FPGA are connected to other external component using LVCMOS standard.

[1] Available at `https://www.digikey.be/product-detail/en/xilinx-inc/XC6SLX9-2TQG144C/122-1745-ND/2339919`

The FPGA are programming using a classical JTAG chain [46] using the Xilinx Platform Cable [43]. That chain allows to program the 4 FPGA at the same time and is therefore user-friendly. JTAG is the only way to program the FPGA for simplicity reason as well as debugging efficiency.

Security issues According to the security issue exposed in section 2.1, the slave FPGA should not be connected in any wired way. To ensure the slaves do not exchange information without going through the master, the power domains of each FPGA are split, and the JTAG chain can be unconnected by removing jumpers after programming. This way, the slave cannot recover the secret value and trigger a cheat-code trojan.

External Components In addition to the previously exposed characteristic, the FPGAs are also connected to other components. First each of the slave is connected to 4 LEDs and a standard 26 Pins header: this allows additional connections efficient for debugging and performances analysis. All the FPGAs can be connected to an UART transceiver that allows communication with any personal computer. The reset signal of the FPGA is based on an external push-button, but could also be forwarded by the master. The internal clock needs to be externally generated: a waveform generator is connected to the master, which forwards that clock to the slaves. A PLL can be used to increase the frequency of the internal clock.

Component	Description
Internal Logic Power Supply	1.2 V
I/O Power Supply	3.3 V
Master Com. LVCMOS	24 pins in slave 3, 26 pins in slave 1-2
Master Com. LVDS	12 With slave 3, 13 pins slave 2, not available slave 1
Programming	JTAG chain
External Com.	UART Transceiver connection via jumper
CLOCK	External with possibly internal PLL
LEDS	4
Header	26 pins
Reset	External button

Table 4.1: FPGAs main characteristics description

Errata Some issue were found while using the PCB. Firstly, two pins involved in the JTAG programming where exchanged: thanks to the jumpers placed for debugging and security issues, that error was easily corrected. Secondly, two differentials pins also involved in programming were exchanged, resulting in negated control signals entering all the FPGAs. To fix this, a second small PCB was built to pre-invert the negated signals.

4.2 Results and measurements

In order to measure the performances of a real physical implementation of a cryptographic system, the metric of throughput is used. This is defined as the number of bits encrypted per second: in the chapter, its unit is the Megabit per second denoted Mbps. The goal is to reach a maximum throughput by keeping the correctness of the system. In order to validate those two, two dedicated pins are used on the master header. The first one is rising at each encryption, while the second one is raised if one or more encryptions failed.

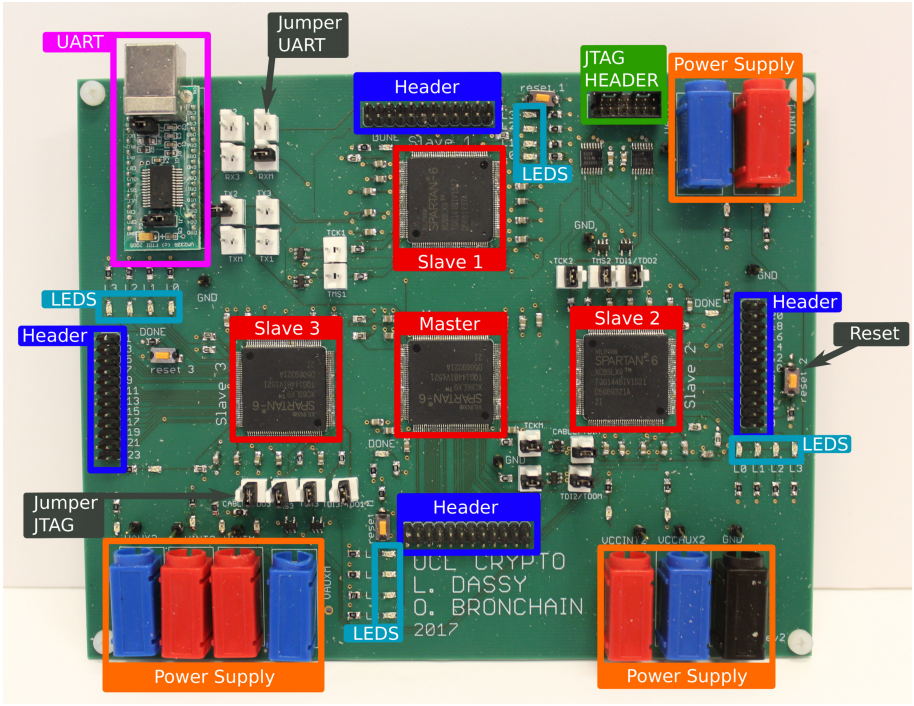


Figure 4.1: The circuit board.

4.2.1 Experimental setup

The measurement setup presented in Figure 4.2 is composed of different elements. First the FPGA are programmed using the PC and the programming cable: the PC is used to write and test HDL (Hardware Description Language). This is translated, mapped, and used by the programming cable to configure the FPGA on the demonstration board. The platform cable and the board are connected to the board using the JTAG header.

The clock generator is connected to the master header: it uses that clock to feed its PLL. The slaves receive their clock from the master. The DC power supply generates the internal and I/O supply voltages. For simplicity, the slaves are directly connected to the DC supply, even though they could use it to break the hypothesis of no direct slave communication. For a real system, the power supplies should be separated. Finally, the oscilloscope is used to measure the actual throughput of the system. It is connected to the header of the master, on a pin that is raised at each encryption. By observing the frequency of that signal and multiplying it by the plaintext size, the throughput is obtained.

4.2.2 Demonstration board performances

First the bus frequency shown in Figure 4.3a corresponds to the throughput obtained for a single wire bus used to implement Mysterion block cipher. The internal frequency of all parties is fixed as 25 MHz with varying f_{bus}. As observed in previous sections, by increasing the bus frequency between the master and slaves, the communication throughput increases as well as the encryption throughput. The maximum bus frequency that leads to correct encryption is of 12.5 MHz that corresponds to a factor β equal to 0.5. Note that the improvement in term of throughput is linear with the increasing bus frequency. The maximum throughput obtained on a single wire is around 0.76 Mbps.

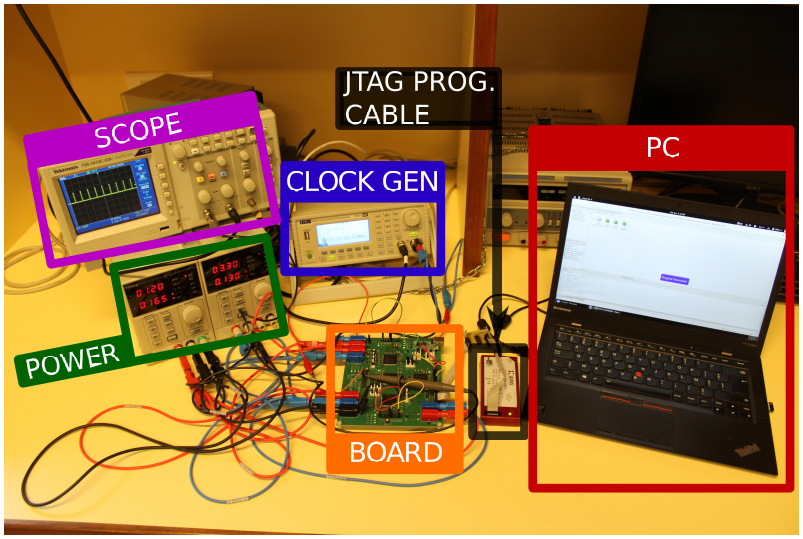
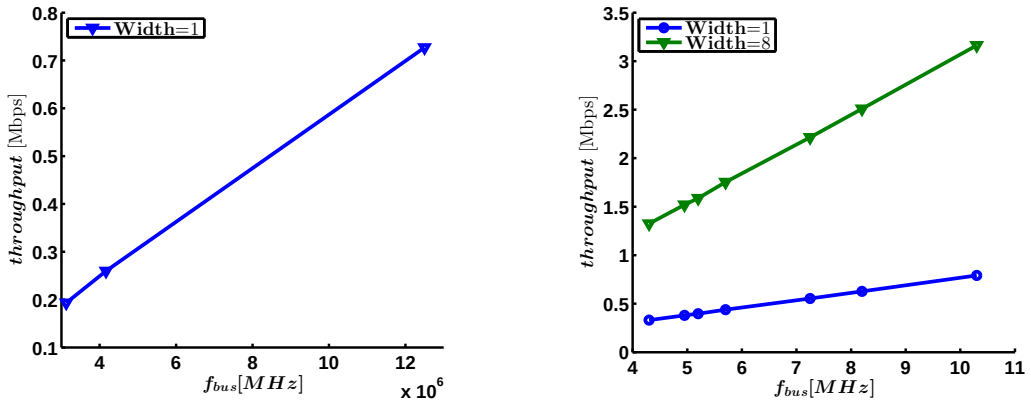


Figure 4.2: Experimental setup for Hardware-Trojan Resilient demonstration board.



(a) Bus frequency f_{bus} influence on throughput for fixed $f_{\text{int}} = 25\,\text{MHz}$

(b) Bus width influence for $f_{\text{bus}} = f_{\text{int}}/2$ and variable f_{int}

Figure 4.3: Parameter influence on performances measured on the Hardware-Trojan resilient demonstration board

Figure 4.3b shows the influence of the width parameter influence versus all the frequencies increasing. Intuitively, the number of cycle is the same since the frequencies are increased, the period of a cycle decreases and therefore, the throughput increases. In that setting, the frequencies are set to $f_{\text{bus}} = f_{\text{int}}/4$ and lead to a maximum throughput of 3.1 Mbps, with $f_{\text{int}} = 40\,\text{MHz}$ for a bus width of 8. This corresponds to a delay of 0.04 ms from input to output of the Hardware-Trojan resilient block cipher. Note that the gain obtained by parallelizing the bus is more important than gain obtained by increasing the bus frequency on a single wire. Indeed, the maximum frequency of a bus of width 8 is around 10 MHz, while it is of 12.5 MHz in the case of a single wire bus.

In the same setting, i.e. $\beta = 0.25 \cdot 8 = 2$, using the number of cycles of the simulation in section 3.5 (Figure 3.10a), AES takes two times more cycles than Mysterion. This results in an estimated throughput of 1.55 Mbps.

4.3 FPGA and utilization

The utilization of the FPGA, i.e. slice, register and LUT occupation are given in Table 4.2 and 4.3 as a function of the bus width. A slice[44] contains 4 LUT (6-bit inputs) and 8 registers. The Spartan 6 LX9 has a total of 1430 slices, but since they are not fully utilized, the slice usage percentage is higher than max $\left(\frac{\text{LUT}}{4}, \frac{\text{Register}}{8}\right)$.

Bus Width [bits]	**Slices**	**LUT**	**Registers**
1	36 (2.5%)	72 (1.2%)	59 (0.5%)
4	38 (2.6%)	78 (1.4%)	59 (0.5%)
8	47 (3.2%)	92 (1.6%)	59 (0.5%)

Table 4.2: Master FPGA resources utilization. It includes a module for serial communication, a PLL, and clock distribution to the slaves.

Bus Width [bits]	**Slices**	**LUT**	**Registers**
1	1417 (**99%**)	4729 (82%)	2893 (25%)
8	1382 (**96%**)	4637 (**81%**)	2893 (25%)

Table 4.3: Slave FPGA resources utilization. The internal correlated randomness generation is done with two 55 cycles AES implementations.

The two internal AES use roughly 25% of the slave LUT and 10% of the registers, for a total of around 30% of the slices. To shrink the slave implementation, another time-space trade-off could be used, with only one internal AES block used in series, or by using a lighter cipher. This also shows that the size of the master is small compared to a single typical AES: around 8 times smaller in term of slices. This means that the number of gates to trust is at least 8 times lower in the suggested HT resilient architecture compared to classical AES implementation.

4.4 Performance summary

The performances obtained on the demonstration board are summed up in Table 4.4 and demonstrates the feasibility of a Hardware-Trojan resilient architecture. In that sens, the open question of the performances and bottlenecks of such a system is answered: even with non ideal bus communication, the maximum throughput of 3.1 Mbps already allows different range of applications as discussed hereunder.

Block ciphers comparison As mentioned earlier, two block ciphers were simulated, namely AES and Mysterion but only the second one was physically tested on the demonstration board. However, the interpolation based on simulations results is straightforward and is shown in Table 4.4.

The results of the two block ciphers are comparable on different points: by increasing the bus width, the number of cycles for an encryption decreases and implies a throughput benefit as mentioned in subsection 3.5.3. Since the two block ciphers are supposed to be implemented on the same physical board and that the intrinsic performances of the PCB does not depends on the block cipher running on the FPGAs, the maximum frequency obtained is the same.

The cost of implementing either AES or Mysterion is sensibly the same. In the two cases, the FPGAs are nearly fully used. There is no significant difference in the amount of logic for the different slaves. Therefore, a conclusion is that the price of those implementations is similar and thanks to the demonstration board, it is shown that a triplet can be implemented with around

Bus settings		Mysterion		AES	
Width	**Frequency**	**Cycles**	**Throughput**	**Cycles**	**Throughput**
1	**12.5** MHz	8387	0.76 Mbps	16820	0.38 Mbps
2	**10** MHz	4547	1.1 Mbps	9119	0.55 Mbps
4	**10** MHz	2627	1.9 Mbps	5268	1.0 Mbps
8	**10** MHz	1667	3.1 Mbps	3343	1.55 Mbps

Table 4.4: Best performances in terms of cycles and throughput obtained with different bus width for AES and Mysterion block ciphers ($\beta' = 0.25$)

100 euros. A discussion about the trade-off between cost and security is proposed in the conclusion.

Those block ciphers differ by their throughput as shown in Table 4.4: Mysterion beats AES by a factor two. This is due to the intrinsic properties of Mysterion that is designed to minimize the non linear operations that require data transfer. However, AES has the advantage of being a world battle block cipher while Mysterion is still a research tool.

Application Notes This demonstration board highlights the possibility of building a Hardware-Trojan resilient architecture with a sufficient throughput for low-bandwidth application. However, in that case the architecture is not ideally designed. Due to the hypothesis, the number of tests t must be significantly higher than the number of real runs n. This implies that if the test and the real runs are performed at full throughput (3.1 Mbps), the testing phase must also be significantly longer than the real runs phase. An improvement would be the possibility to mix the testing phase with the real run phase. In that case, one part of the total throughput would be dedicated to verification while the other would serve the real runs. However, this would require a review of the framework.

A less demanding type of application corresponds to the case where it does not require the full bandwidth, but a fixed number of runs. It was found here above that $24 \cdot 10^3$ encryptions are performed in one second, for a device that needs to perform 10^6 runs, with a ratio $t/n = 100$, this implies 10^8 tests can be performed in one hour, which is reasonable. A good enough ($100^{\frac{\lambda}{2}}$) security level could be selected by the amount of hardware used λ. Such a device could for example be a chip controlling a lock mechanism: supposing that it encrypts a plain text every minute, its security is guaranteed for more than 1000 years with an hour of tests.

Extrapolation Table 4.5 shows a summary of the performance of our implementation, and extrapolates them to a better performing bus like PCI and a wider SPI. To use a wider SPI, FPGAs with more I/O available should be used. The performance could probably be even better with differential I/O and a large FPGA package for the master. Since the internal frequency of the slaves is quite low in our implementation, we consider that the internal frequency can scale linearly.

For a PCI Express 3.0 bus, more expensive FPGAs are required, but the bus throughput and internal frequency can be dramatically increased. This cost is due to the fact that the master FPGA must handle the forwarding between three PCI express links. With the current implementation, Mysterion would reach 208 Mbps, and AES 105 Mbps. When the linear part of the block ciphers is neglected, the higher bound is 5.2 Gbps for Mysterion, and 2.5 Gbps for AES. This bound could be approached by enhancing the implementation of the linear part, especially given that those FPGAs are larger, and that using this area for faster linear implementation is possible.

Communication Bus				Block Cipher		Cost
Technology	**Width**	**Freq.**	**Throughput**	**AES**	**Mysterion**	
Custom SPI	8	10 MHz	80 Mbps	1.55 Mbps	3.1 Mbps	€140 $\cdot\lambda$
Custom SPI	32	10 MHz	320 Mbps	6.2 Mbps	12.4 Mbps	€800 $\cdot\lambda$
PCI Express 3.0	8	8 GHz	63 Gbps	105 Mbps, up to 2.5 Gbps	208 Mbps, up to 5.2 Gbps	€15,000 $\cdot\lambda$

Table 4.5: Summary of the achieved block ciphers performances, and extrapolation to a 32 bits SPI and a PCI Express 3.0 x8 bus.

5

Side Channel Analysis applied to Hypothesis Verification

With the architecture proposed in previous chapters, it was shown that reducing the amount of trusted gates to a very restricted number is possible. By exploiting that behavior, it is believed that classical side-channel detection [29, 1, 30] would work more convincingly. Indeed, it is commonly assumed and quite intuitive that by decreasing the size of an IC, the side-channel detection of an HT is easier since the footprint and impact of the malicious gates increases.

In this chapter, the special case of "time bomb" trojans detection inside the master is inspected by offering a new methodology based on classical side-channel analysis tools. First, a leakage model for the master based on [30] is suggested, followed by a new technique for counter detection based on side-channel analysis. Finally, the experimental results for the detection of that type of trojan inside the master are discussed.

5.1 Leakage modeling

The leakage is defined as the information that can be recovered by measuring physical phenomena related to the circuit, e.g. electromagnetic emissions due to the internal currents, or power consumption. It also encompasses side effects of the implementation, like timing differences in the output. In this chapter, when leakage is mentioned, it is assumed to be the power consumption.

Typically, the leakage of a genuine circuit $L_g()$ is modeled based on Eq. 5.1, and is composed of three terms [30, 35]. First $\eta\left(t;\sigma^2\right)$ is the noise term: it includes process and measurement noise. They are modeled as additive white Gaussian noise: therefore η is Gaussian distributed, with mean zero and variance σ^2. It is assumed to be data independent and stationary, as it should not depend on previous realization of the stochastic process. The subscript g represents the genuine term of an IC.

$$L_g\left(t;I;O;\sigma^2\right) = \tau_g(t) + \beta_g(t;I;O) + \eta\left(t;\sigma^2\right) \tag{5.1}$$

The term $\beta_g(t;I;O)$ is directly depending on the input I and the output O of the circuit. This implies that $\beta_g()$ is not stationary and that the leakage of a circuit depends on its previous inputs. Finally, $\tau_g(t)$ is the part of the leakage that does not depend on the input of the circuits. It models the leakage of the control gates, since those are data independent.

An Hardware-Trojan leakage L_T presented in Eq. 5.2 is the leakage of a malicious IC implementing the functionality of the previously described genuine circuit. L_T includes two additional terms that model the gates introduced by the HT. First $\tau_T(t)$ is the leakage from the gates that are independent of the input, and $\beta_T(t;I;O)$ the leakage of the dependent gates.

$$L_T\left(t;I;O;\sigma^2\right) = \tau_g(t) + \tau_T(t) + \beta_g(t;I;O) + \beta_T(t;I;O) + \eta\left(t;\sigma^2\right) \tag{5.2}$$

The model proposed above is simple since it does not take into account the process variations, i.e. the static power consumption variation between two different circuits. Such a behavior is increasing with smaller technology nodes. The main challenge is to differentiate between genuine (Eq. 5.1) and trojanized (Eq. 5.2) leakage in the presence of process variation. Indeed, a small enough Hardware-Trojan could be hidden in the process variation and therefore never be detected.

5.2 Counter detection

In this section, the case study of Hardware-Trojan based on a counter inserted in the master is described. Since a counter is independent of the input, the term $\beta_T(t;I;O)$ is equal to zero and Eq. 5.2 is therefore rewritten as $L_C()$ in Eq. 5.3.

$$L_C\left(t;I;O;\sigma^2\right) = \tau_g(t) + \tau_C(t) + \beta_g(t;I;O) + \eta\left(t;\sigma^2\right) \tag{5.3}$$

This section first lays out a side-channel based methodology to detect counter, then describes an experimental setup implementing the methodology, and finally shows the obtained results.

5.2.1 Methodology

In order to detect an Hardware-Trojan, the framework presented in Figure 5.1 is used. This proposed methodology allows to differentiate a Trojan infected and genuine circuit, and requires two different IC.

1. *Golden IC*: is supposed to be implemented honestly and therefore has a leakage corresponding to Eq. 5.1.
2. *Candidate IC*: is either malicious or honest. The goal is to classify it in one of the two classes. The classification is performed by comparing the leakage since in the first case it should correspond to Eq. 5.3, while in the second case, to Eq. 5.1.

This methodology consists in three distinct steps: firstly, measurements are taken on the golden IC, and features are extracted from those. Then the same measurements are taken on the candidate IC and features are extracted in the same way. Finally, a comparison is performed between the features to classify the candidate IC in the genuine or infected class. Those steps are detailed hereunder.

Measurements The first step is to take N samples of a leakage on a candidate IC $L_{\text{IC}}()$ and to average those: for each measurement, the inputs of the IC are set. That average is denoted as $\overline{L_{\text{IC}}}(t)$ in Eq. 5.4. The goal of such an averaging is to get rid of the noise term $\eta\left(t;\sigma^2\right)$.

$$\begin{aligned}\overline{L_{\text{IC}}}(t,I) &= \frac{1}{N}\sum_{i=1}^{N} L_{\text{IC}}(t;I;O;\sigma^2) &(5.4)\\ &= \tau_{\text{IC}}(t) + \beta_{\text{IC}}(t;I;O) + \frac{1}{N}\sum_{i=1}^{N}\eta(t;\sigma^2)\\ &\simeq \tau_{\text{IC}}(t) + \beta_{\text{IC}}(t;I;O) &(5.5)\end{aligned}$$

For N sufficiently large, the noise term $\eta(t;\sigma^2)$ can be removed since it has a mean of zero. Since $\tau_{\text{IC}}(t)$ is deterministic and is common between the genuine and the trojanized circuit, it can be removed from the sum. Note that $\beta_{\text{IC}}()$ is a deterministic function: since the inputs are the same for all the measurements, it can also be removed of the sum in Eq. 5.5.

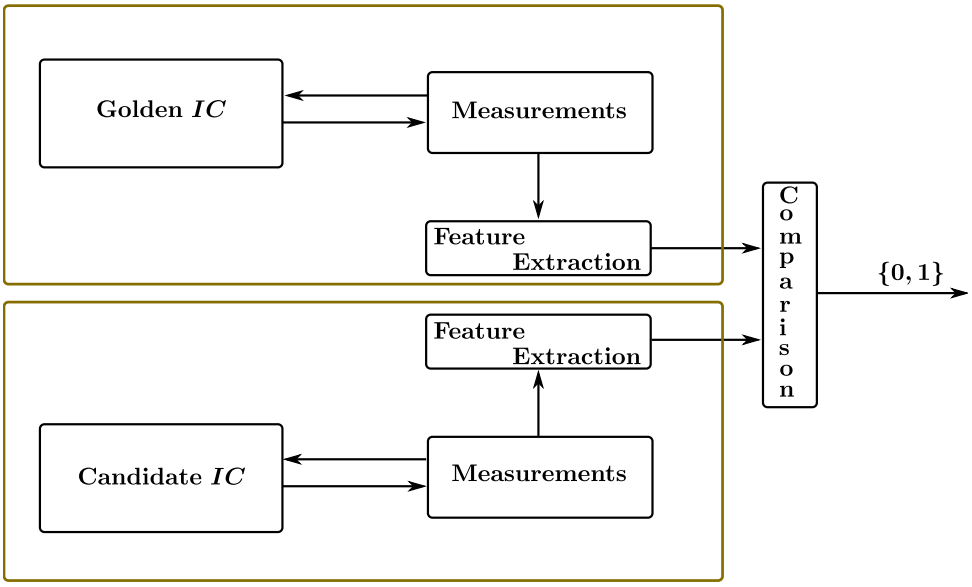


Figure 5.1: Framework for Hardware-Trojan detection from [15].

Feature extraction In the case of a golden circuit, the term $\tau_{\text{IC}}(t)$ in Eq. 5.5 is equal to $\tau_g(t)$, while in the case of an infected IC, it is equal to $\tau_g(t) + \tau_c(t)$. The goal is therefore to find a criterion that allows to differentiate between the two cases. The average difference is one of the possibilities, as shown in Eq. 5.6. For notation simplicity, output O variables are removed since it directly depends on the inputs in the case of a deterministic circuit.

$$\overline{L_{\text{GOLD}}}(t, I) - \overline{L_{\text{IC}}}(t, I) = \tau_g(t) - \tau_{\text{IC}}(t) + \beta_g(t, I) - \beta_{\text{IC}}(t, I) \tag{5.6}$$

$$= \tau_g(t) - \tau_{\text{IC}}(t) \triangleq \delta(t) \tag{5.7}$$

In that case, if IC is genuine, Eq. 5.7 is equal to zero. In the case of an infected IC, the only term remaining is $\tau_c(t)$. Intuitively, since $\tau_c(t)$ represents the leakage of a counter that is increased at each clock cycle, it should have a high signature in the frequency domain. Therefore, the selected criterion is the Fourier transform of the difference $\delta(t)$, denoted $\mathcal{F}(\delta(t)) = \Delta(f)$.

A first feature extraction method is proposed in Algorithm 13. That feature extraction method takes as input the power traces $\overline{L}$ of a golden reference and the traces $\overline{L_{IC}}$ of a candidate IC that needs to be classified. Each $\overline{L}$ corresponds to a given set of inputs I. The first step is to compute $\Delta(f, i)$ for each different input $i \in I$. Then the algorithm looks for a *point of interest*. This is done by searching for the frequency with the largest difference of mean. The output is then the samples with the largest difference compared to the golden reference, at the specific frequency f_{POI}. Each sample of the output corresponds to an input $i \in I$.

Algorithm 13: Feature extraction 1

Data: $\overline{L_{\text{GOLD}}}(t, i)$ and $\overline{L_{IC}}(t, i)\ \forall i \in I$
Result: $D(i)$, the coefficient at frequency f_{POI} for each input $i \in I$ of the IC.
for $i \in I$ **do**
  $\Delta(f, i) \triangleq \mathcal{F}\left(\overline{L_{\text{GOLD}}}(t, i) - \overline{L_{\text{IC}}}(t, i)\right)$
$f_{\text{POI}} \triangleq \arg\max_f \left(E\left[\Delta(f, i)\right]\right)$
$D(i) \triangleq \Delta(f_{\text{POI}}, i)\ \forall i \in I$

A second feature extraction is proposed in Algorithm 14, which follows the same steps as the previously explained method. The difference is that, as in [31], the output contains the sum of the differences across all frequencies.

Algorithm 14: Feature extraction 2

Data: $\overline{L_{\text{GOLD}}}(t,i)$ and $\overline{L_{IC}}(t,i)\ \forall i \in I$
Result: $D(i)$, the coefficient summed across all frequencies, for each input $i \in I$ of the IC.
for $i \in I$ **do**
  $\Delta(f,i) \triangleq \mathcal{F}\left(\overline{L_{\text{GOLD}}}(t,i) - \overline{L_{\text{IC}}}(t,i)\right)$
$D(i) \triangleq \sum_{f \in F} \Delta(f,i)$ with $i \in I$

Comparison The last step in the framework presented in Figure 5.1 is to differentiate between a genuine or Trojan circuit. The goal is to differentiate the distributions obtained from the feature extraction algorithm when passing a good or Trojan IC. If the two distributions are similar, the candidate is defined as a genuine circuit, otherwise as a Trojan one.

A solution proposed in [15] is to use machine learning tools: however those techniques are quite expensive compared to the relative simplicity of a statistical test. A proposition is to use a Welsh's t-test, which tests the hypothesis that the two distributions have equal means [9]. It requires the random variables tested to be normally distributed, with possibly different variance. The null and alternative hypothesis are written in Eq. 5.8 with μ_g (resp. μ_c) the mean of the golden (resp. candidate) distribution. In other words, the null hypothesis is that candidate circuit is genuine, while the alternative is that the candidate is a Trojan.

$$\begin{aligned} H_0 &: \ \mu_g - \mu_c = 0 \\ H_1 &: \ \mu_g - \mu_c \neq 0 \end{aligned} \tag{5.8}$$

Intuitively, the mean of a genuine distribution should be equal to zero since the single term remaining in Eq. 5.7 should be white Gaussian noise $\eta(t,\sigma^2)$. Therefore, the mean of the golden circuit μ_g in statistical hypothesis Eq. 5.8 could be set to zero. One may also choose a threshold of the p-value returned by the statistical test to determine if the IC is genuine or not.

5.2.2 Measurement setup

The experimental setup is shown in Figure 5.2: it includes two FPGAs communicating together. The first one (Control) is sending an input $i \leftarrow I$ with I a set of inputs to the second one (Target). That Target FPGA contains the master described previously. Those two FPGAs are connected on a *Sakura-G* side-channel demonstration board [32]. The leakage is measured on a small resistor in series with the Target FPGA power network. That measure is amplified by an on-chip amplifier.

The technical details of the setup are shown in Table 5.1. The first inserted Trojan is using around 12% of the original golden master resources, which is reasonable since the master should be very small. The second is smaller and only increases the total IC size by 4%. Note that the Target FPGA does not only contain the master, but also some circuit to trigger the oscilloscope.

In that setup, three different masters are measured: a *golden*, a *genuine*, and a *trojan*. Precisely, *golden* and *genuine* have exactly the same layout since no re-synthesis has been performed. *Trojan* is the same as *golden* but with an additional 128 or 32 bits counter. That counter increases at each clock cycle. Since that netlist is re-synthesized, the layout of *Trojan* may be different from *golden* even for the common gates.

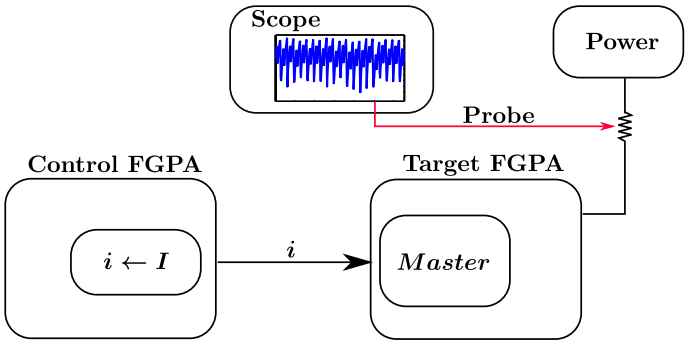


Figure 5.2: Measurement Set Up for HT detection.

Setting	Description
Board	*Sakura-G* [32]
Target FPGA	Xilinx Spartan-6 XC6SLX75
Control FPGA	Xilinx Spartan-6 XC6SLX9
Bus description	Width = 1 and Frequency = 1.5 MHz
Oscilloscope	LeCroy, 50 [MSample/s] with 12 bits precision
Operation frequency	3 MHz
Input set	All integers between 0 and 127. Encoded on 7 bits
Measurements	N = 500 on a 1Ω resistor
Trojan 1	128 bits counter that increases of 12% the LUTs and 15% the registers
Trojan 2	32 bits counter that increases of 3.4% the LUTs and 4.1% the registers

Table 5.1: Experimental setup description

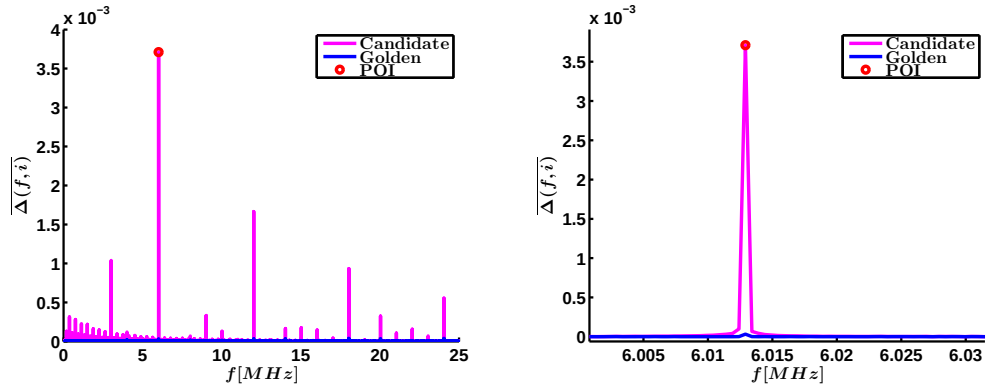
5.2.3 Experimental results

This section discusses the results of the distribution output by the feature extraction methods proposed previously. The solution using Fourier analysis is compared to the standard methodologies operating in time domain. Hereunder, all the blue (resp. magenta and green) lines corresponds to the genuine (resp. Trojan) IC.

Feature Extraction Figure 5.3 shows the average of $\mathcal{F}(\delta(t))$ on the set of input I. This average is exploited by Algorithm 13 and 14. On the entire spectrum (Figure 5.3a), one may notice that the significant difference are located around the frequencies multiple of f_{int} and f_{bus}.

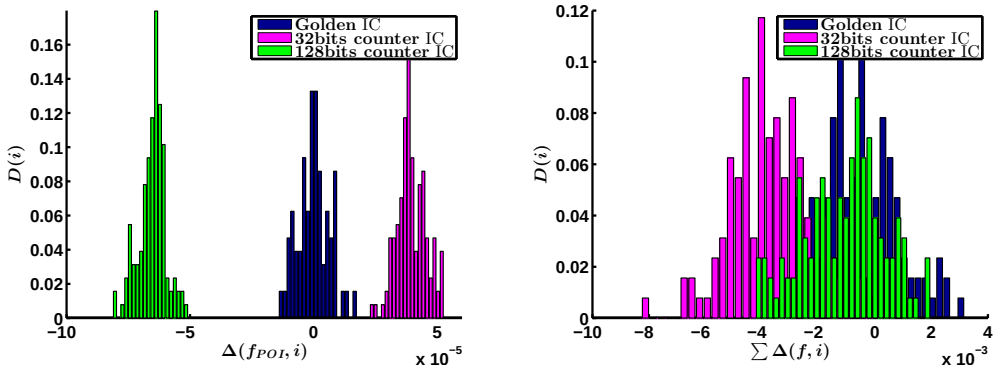
The $\Delta(f)$ is shown around the point of interest (POI) in Figure 5.3b, the POI being the frequency where the $\Delta(f)$ mean is the largest, meaning that it is the most distant to the *golden* reference. One observes that the mean of $\Delta(f)$ is significantly different between *genuine* and *trojan*.

This observation is confirmed by plotting the distribution output by Algorithm 13: in Figure 5.4a, the distribution corresponding to a *genuine* circuit is significantly different from the one including a trojan. That difference is smaller with the Algorithm 14, as shown in Figure 5.4b.



(a) Entire spectrum **(b)** Spectrum around POI

Figure 5.3: Mean of $\mathcal{F}(\delta(t)) = \Delta(f)$ on the set of input I for *genuine* and *Trojan* IC.



(a) Distributions obtained by performing Algorithm 13 **(b)** Distributions obtained by performing Algorithm 14

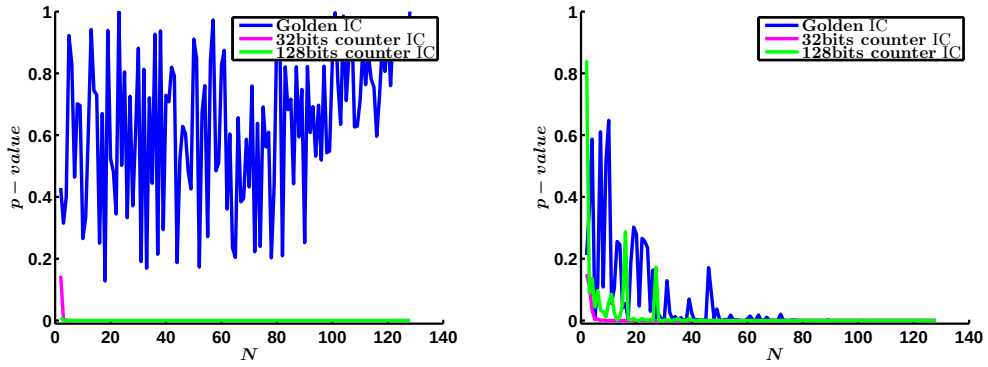
Figure 5.4: Distributions outputted by various feature extraction algorithms

Comparison Once the distributions representing the ICs are extracted, the comparison phase is performed. As mentioned above, a t-test is used to perform the comparison with the null hypothesis of zero mean.

In Figure 5.5, the p-value of the previously described statistical test is shown for N random samples taken from the corresponding distributions in Figure 5.4. N also denotes the size of the input set I that is used to classify an IC since an input i corresponds to a distribution sample.

From Figure 5.5, we see that the statistical test gives significantly different p-value depending on the IC type for the first proposed algorithm, while the second one does not, since it does not respect the zero-mean hypothesis for genuine reference. Algorithm 13 is performing better compared to Algorithm 14 since it requires around 4 samples to detect a Trojan while the alternative does not match the hypothesis.

The number of samples used to detect the Trojan is very low and is therefore promising for future works: the number of samples required to distinguish a trojan circuit compared to its relative size is a key point in trojan detection. The intuition is confirmed by observing that more traces are required to detect a 32 bits counter trojan than for a 128 bits counter.

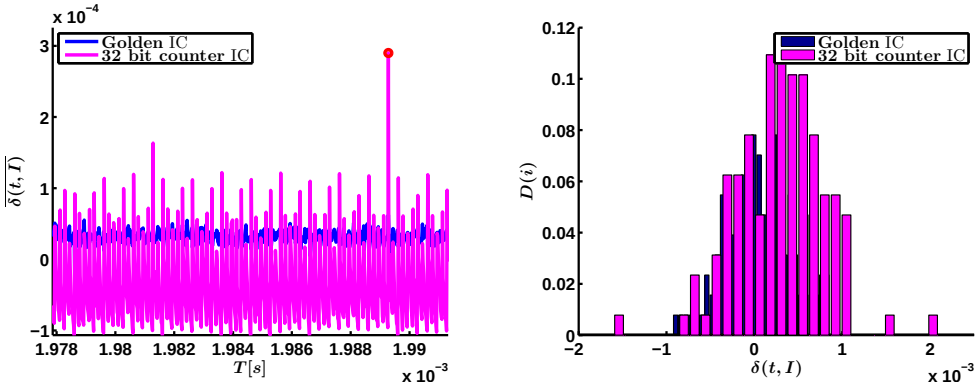


(a) for distribution from Figure 5.4a (Algorithm 13)

(b) for distribution from Figure 5.4b (Algorithm 14)

Figure 5.5: T-test performed for distributions of feature extraction algorithm for various number of sample size

Benefits of the proposed method The results shows that using a Fourier analysis to perform the detection of a counter is sound. Indeed, by applying the same methodology as Algorithm 13 in time domain, the two distributions in Figure 5.6a are visually similar, and it is hard to distinguish them. An increased number of traces could probably help to differentiate those.



(a) Time series

(b) Distribution at POI

Figure 5.6: Mean of $\delta(t)$ on the set of input I for *genuine* and *trojan* IC.

Discussion Thanks to experimental results, one key point is highlighted: reducing the master size allows to perform efficient side-channel detection. This is due to the increased trojan impact on side-channel information compared to the rest of the IC. The Hardware-Trojan resilient architecture presented in chapter 3 is therefore meaningful, in the sense that the detection problem becomes smaller when trying to detect a Trojan only in the master.

In this chapter, a specific type of inserted trojan is addressed, namely counters increased at each clock cycle. An interesting topic would be to observe the influence on that detection methodology if the counter is increased at another frequency: is the difference as clear as in this chapter? How is the point of interest changing with a different counter frequency?

The case of “cheat code” trojan using multiplexer is left for future work, but some questions need to be raised: is this methodology efficient even if the power consumption is not as regular as for counter based trojan? Would that variation in power consumption result in a significantly modified spectrum? Would a multiplexer trojan be detectable if it is not triggered by the set of inputs?

Conclusion

Trojan resilience is the ultimate security goal, where an adversary has unlimited power: indeed, he can modify the implementation of the system. This type of attack is particularly hard to detect, and countermeasures are truly needed since most of the IC fabrication process happens abroad. In that context, a new framework presented at ACM CCS 2016 has been proposed to mitigate that type of attacks [11].

Contributions

That new theoretical solution, based on multi-party computation, is implemented in this work answering the initial open questions. The viability of that solution needed to be shown as well as the possibility of implementing a small sized trusted master reducing the amount of gates that needs to be trusted.

Two block ciphers have been implemented within that Hardware-Trojan resilient framework, namely AES and Mysterion. AES is used worldwide and recommended by the U.S. government, while Mysterion is a recent lightweight research cipher suitable for multi-party computation. It has the benefit of an efficient representation, but also of an improved multi-party computation protocol that drastically reduces the communication requirements.

Based on those improvements, a new architecture has been designed and build on a demonstration board connecting four FPGAs. This allows us to evaluate the performances of such a Trojan-resilient circuit on a concrete basis, to confirm practical applicability, and to identify bottlenecks and tracks for improvement. Concretely, performances measurement shows that a throughput of 1.55 Mbps AES and 3.1 Mbps Mysterion encryption can be obtained even with low cost components and a significantly reduced amount of trusted gates.

That limited number of trusted gates allows to apply the traditional detection techniques based on side-channel analysis resulting in convincing results. For all side-channel detection techniques, the results are viable up to a given trojan size compared to the rest of the circuit. Since the master is small, the side-channel detection remains efficient even for small malicious modifications.

Impact

Concretely, this work already proposes solutions to real life problems: for critical applications like diplomatic conversations between two distant embassies, the integrated circuits supporting the encryption are currently purchased abroad. The only security guarantee relies on the trust

in the foreign chip manufacturer. The lack of financial resources forces to rely on weak security proofs or even no proof at all. The performances obtained with the demonstration board for Hardware-Trojan resilient implementation would allow those critical organization with low funding to ensure that their physical devices are secure even if they are purchased abroad.

In the concrete example of a one hour phone call at 64 kbps between two sensible remote interlocutors, the demonstration board offers a convincing and applicable alternative (Table 5.2). Even with one single day or single night of required tests, the one hour conversation can be reliable in the trade of more hardware usage.

Thanks to an exponential decrease of security bound with the amount of independent circuit, by slightly increasing the number of chips, the security guarantees are significantly improved. Even with 20 independent triplets which represents at total cost of 2000 euro, embassies can ensure the security of their physical devices implementing block ciphers against Hardware-Trojan attacks.

λ	**Cost** [€]	**Testing** [hour]	**Prob. of HT attack**	
			Mysterion	**AES**
2	200	12	10^{-3}	10^{-3}
		24	10^{-4}	10^{-3}
10	1000	12	10^{-14}	10^{-13}
		24	10^{-16}	10^{-14}
20	2000	12	10^{-28}	10^{-25}
		24	10^{-31}	10^{-28}

Table 5.2: Cost and Security of one hour phone call with the demonstration board.

Glossary

AES	Worldwide used block cipher standardized by NIST in 2001 based on Galois Field (GF) arithmetic. AES $GF\left(2^8\right)$ denotes that the S-box inversion is naively implemented while AES $GF\left((2^4)^2\right)$ is optimized for MPC protocol..
Block Cipher	Keyed pseudo random permutation used to protect data based on the knowledge of the encryption key. Used in symmetric cryptography with examples as AES and Mysterion.
Cipher Text	Defined as the encrypted message obtained by running cryptographic function with a given plain text.
Correlated Random	A set of element uniformly distribution with their addition $\oplus$ is equal to zero. Building block of the MPC protocol.
FPGA	Programmable device programmed with logic netlist, gates used to validate implemented architecture.
Galois Field	A field that contains a finite number of elements represented as polynomial. The Galois Field $GF\left(2^8\right)$ is the building arithmetic of AES.
Golden	Denote a trusted reference that can be compared with a candidate to determine if it can be trusted.
Hardware-Trojan	Any IC that does not match its desired specification due to the modification of an attacker $\mathcal{A}$. This can introduce in example information leakage or denial of service.
HDL	Programming language that describe logic gates interactions. For this work, Verilog was used.
Input Scrambling	Making the input of a device distributed uniformly at random i.e. thanks to the usage of an MPC protocol.
Master	Master trusted circuit of the HT architecture. Should include a minimal number of gates to verification facilities.
MixColumns	Building block of AES involving only constant multiplication in Galois Field and is so linear. Part of the diffusion layer.
MPC	Cryptography branch that allows different parties to compute a function $f(x)$ without having any information on x.
Mysterion	Light weighted block cipher initially designed for efficient masking properties and so efficient in MPC protocol.
Netlist	File describing logic gates connection derived from an HDL language by a CAD tool.
Plain Text	Defined as the clear message that must be encrypted by a cryptographic function resulting in the cipher text.
S-Box	Non linear part of a block cipher that introduces confusion, the most resource intensive part of a block cipher.
ShiftColumns	Building block of Mysterion involving only wire crossing and is so linear. Part of the diffusion layer.

ShiftRows	Building block of AES involving only wire crossing and is so linear. Part of the diffusion layer.
Side Channel	All the information leaking from an electronic device. In this work, this denotes most of the time power consumption.
Slave	Slave untrusted circuit of the HT architecture. Considered as a black box that is tested t times before usage.
SPI	Serial Peripheral Interface, used to connect the slaves to the master.
Stationary process	A stochastic process where the probability function does not change when shifted in time.
Xilinx	FPGA manufacturer that provides Spartan 6 LX9 chips used to build the demonstration board.

Acronyms

AES	Advanced Encryption Standard.
CAD	Computer-Aided Design.
CLK	Clock.
DC	Direct Current.
FPGA	Field-Programmable Gate Array.
GF	Galois Field.
HDL	Hardware Description Language.
HT	Hardware Trojan.
I/O	Input/Output.
IC	Integrated Circuit.
IoT	Internet of Things.
IP	Intellectual Property.
JTAG	Joint Test Action Group.
LED	Light-Emitting Diode.
LUT	Look-Up Table.
LVCMOS	Low Voltage Complementary Metal Oxide Semiconductor.
LVDS	Low Voltage Differential Signaling.
MISO	Master Input Slave Output.
MOSI	Master Output Slave Input.
MPC	Multi-Party Computation.
nSS	not Slave Select.
PCB	Printed Circuit Board.
PCIe	Peripheral Component Interconnect Express.
PLL	Phase Locked Loop.
POI	Point of Interest.
PPT	Probabilistic Polynomial Time.
PRNG	Pseudo-Random Number Generator.
RNG	Random Number Generator.
SEM	Scanning Electron Microscope.
SPI	Serial Peripheral Interface.
TQPF	Thin Quad Flat Package.
TRNG	True Random Number Generator.
UART	Universal Asynchronous Receiver Transmitter.
VLSI	Very Large Scale Integration.

Bibliography

[1] Dakshi Agrawal, Selçuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan detection using ic fingerprinting. In *IEEE Symposium on Security and Privacy*, pages 296–310. IEEE Computer Society, 2007.

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 805–817, New York, NY, USA, 2016. ACM.

[3] Daniel Augot and Matthieu Finiasz. Direct construction of recursive MDS diffusion layers using shortened BCH codes. *CoRR*, abs/1412.4626, 2014.

[4] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. Hardware trojan attacks: threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.

[5] Rajat Subhra Chakraborty, Francis Wolff, Somnath Paul, Christos Papachristou, and Swarup Bhunia. *MERO: A Statistical Approach for Hardware Trojan Detection*, pages 396–410. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[6] Franck Courbon, Philippe Loubet-Moundi, Jacques J. A. Fournier, and Assia Tria. A high efficiency hardware trojan detection technique based on fast sem imaging. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 788–793, San Jose, CA, USA, 2015. EDA Consortium.

[7] Ronald Cramer and Ivan Damgård. Multiparty computation, an introduction. In *Contemporary cryptology*, pages 41–87. Springer, 2005.

[8] Joan Daemen and Vincent Rijmen. The Rijndael Block Cipher AES proposal. Katholieke Universiteit Leuven, 1999.

[9] François Durvaux and François-Xavier Standaert. From improved leakage detection to the detection of points of interests in leakage traces. Cryptology ePrint Archive, Report 2015/536, 2015. `http://eprint.iacr.org/2015/536`.

[10] François Durvaux, François-Xavier Standaert, and Santos Merino Del Pozo. Towards easy leakage certification: extended version. *Journal of Cryptographic Engineering*, 7:1–19, 6 2017. http://dx.doi.org/10.1007/s13389-017-0150-0.

[11] Stefan Dziembowski, Sebastian Faust, and François-Xavier Standaert. Private circuits III: hardware trojan-resilience via testing amplification. pages 142–153, 2016.

[12] Samaneh Ghandali, Georg T. Becker, Daniel E. Holcomb, and Christof Paar. A design methodology for stealthy parametric trojans and its application to bug attacks. In *CHES*, pages 625–647. Springer, 2016.

[13] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. Ls-designs: Bitslice encryption for efficient masked software implementations. In *FSE*, pages 18–37. Springer, 2014.

[14] Vincent Grosso and François-Xavier Standaert. Masking proofs are tight (and how to exploit it in security evaluations). Cryptology ePrint Archive, Report 2017/116, 2017. `http://eprint.iacr.org/2017/116`.

[15] Ludovic-Henri Gustin, François Durvaux, Stéphanie Kerckhof, François-Xavier Standaert, and Michel Verleysen. Support vector machines for improved ip detection with soft physical hash functions. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 112–128. Springer, 2014.

[16] Syed Kamran Haider, Chenglu Jin, and Marten van Dijk. Advancing the state-of-the-art in hardware trojans design. *CoRR*, abs/1605.08413, 2016.

[17] Howard M. Heys. A tutorial on linear and differential cryptanalysis. *Cryptologia*, 26(3):189–221, July 2002.

[18] Matthew Hicks, Murph Finnicum, Samuel T. King, Milo M. K. Martin, and Jonathan M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. *j-LOGIN*, 35(6):31–41, dec 2010.

[19] Chipworks Inc. Semiconductor manufacturing- reverse engineering of semiconductor components, parts and process.

[20] Anthony Journault, François-Xavier Standaert, and Kerem Varici. Improving the security and efficiency of block ciphers based on ls-designs. *Des. Codes Cryptography*, 82(1-2):495–509, January 2017.

[21] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[22] HeeSeok Kim, Seokhie Hong, , and Jongin Lim. A fast and provably secure higher-order masking of aes s-box. CHES, 2011.

[23] François Koeune and François-Xavier Standaert. A Tutorial on Physical Security and Side-Channel Attacks. In *Foundations of Security Analysis and Design III : FOSAD 2004/2005*, volume 3655 of *Lecture Notes in Computer Science*, pages 78–108, 11 2006.

[24] Gregor Leander, Mohamed Ahmed Abdelraheem, Hoda AlKhzaimi, and Erik Zenner. *A Cryptanalysis of PRINTcipher: The Invariant Subspace Attack*, pages 206–221. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[25] Gregor Leander, Brice Minaud, and Sondre Rønjom. A generic approach to invariant subspace attacks: Cryptanalysis of robin, iscream and zorro. Cryptology ePrint Archive, Report 2015/068, 2015. `http://eprint.iacr.org/2015/068`.

[26] Hua Li and Zachary Friggstad. An efficient architecture for the aes mix columns operation. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 4637–4640. IEEE, 2005.

[27] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. An FPGA Implementation of Rijndael: Trade-offs for side-channel security. In *IFAC Workshop - PDS 2004, Programmable Devices and Systems*, pages 493–498, Cracow,PL, 2004. Elsevier.

[28] Nele Mentens, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. A systematic evaluation of compact hardware implementations for the rijndael s-box. In *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, pages 323–333, 2005.

[29] S. Narasimhan, D. Du, R. S. Chakraborty, S. Paul, F. G. Wolff, C. A. Papachristou, K. Roy, and S. Bhunia. Hardware trojan detection by multiple-parameter side-channel analysis. *IEEE Transactions on Computers*, 62(11):2183–2195, Nov 2013.

[30] X.-T. Ngo, I. Exurville, S. Bhasin, J.-L. Danger, S. Guilley, Z. Najm, J.-B. Rigaud, and B. Robisson. Hardware trojan detection by delay and electromagnetic measurements. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 782–787, San Jose, CA, USA, 2015. EDA Consortium.

[31] Xuan Thuy Ngo, Zakaria Najm, Shivam Bhasin, Sylvain Guilley, and Jean-Luc Danger. Method Taking into Account Process Dispersions to Detect Hardware Trojan Horse by Side-Channel. In *PROOFS: Security Proofs for Embedded Systems 2014*, BUSAN, South Korea, September 2014.

[32] SAKURA Hardware Security Project. Specifications, 2013.

[33] S. S. Sapatnekar. Overcoming variations in nanometer-scale technologies. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(1):5–18, March 2011.

[34] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact rijndael hardware architecture with s-box optimization. In Colin Boyd, editor, *ASIACRYPT*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2001.

[35] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2005.

[36] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal, Vol 28, pp. 656715*, Otober 1949.

[37] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Des. Test*, 27(1):10–25, January 2010.

[38] M. Tehranippor, R. Karri, F. Koushanfar, and M. Potkonjak. Trusthub.

[39] R. Torrance and D. James. Reverse engineering in the semiconductor industry. In *2007 IEEE Custom Integrated Circuits Conference*, pages 429–436, Sept 2007.

[40] Markus Ullrich, Christophe De Canniere, Sebastiaan Indesteege, Ozgül Küçük, Nicky Mouha, and Bart Preneel. Finding optimal bitsliced implementations of 4 χ 4-bit s-boxes.

[41] Adam Waksman and Simha Sethumadhavan. Silencing hardware backdoors. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 49–63. IEEE, 2011.

[42] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. Fanci: Identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 697–708, New York, NY, USA, 2013. ACM.

[43] Xilinx. *Platform Cable USB II datasheet and User Guide*. `https://www.xilinx.com/support/documentation/user_guides/ug384.pdf`.

[44] Xilinx. *Spartan-6 Family Overview*. `https://www.xilinx.com/support/documentation/data_sheets/ds160.pdf`.

[45] Xilinx. *Spartan-6 FPGA Configurable Logic Block User Guide*. https://www.xilinx.com/support/documentation/user_guides/ug384.pdf.

[46] Xilinx. *Spartan-6 FPGA Configuration*. https://www.xilinx.com/support/documentation/user_guides/ug384.pdf.

[47] Xilinx. *Spartan-6 FPGA Packaging and Pinouts Specification*. https://www.xilinx.com/support/documentation/user_guides/ug384.pdf.

[48] Xilinx. *Spartan-6 FPGA Power Management*. https://www.xilinx.com/support/documentation/user_guides/ug384.pdf.

[49] Jie Zhang, Feng Yuan, Lingxiao Wei, Yannan Liu, and Qiang Xu. Veritrust: Verification for hardware trust. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(7):1148–1161, 2015.

[50] Jie Zhang, Feng Yuan, and Qiang Xu. Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 153–166, New York, NY, USA, 2014. ACM.