POLITECNICO DI TORINO

Master Degree course in Computer Engineering

Master Degree Thesis

Analysis of side-channel leakages on cryptographic circuits

Supervisors
Prof. Antonio José Di Scala

Candidate
Lorenzo Giacobbe

Company Supervisor
Security Pattern
Guido Bertoni
Maria Chiara Molteni

Academic Year 2021-2022

Abstract

The security of cryptosystems is usually evaluated using a black-box approach; an adversary can't access the algorithm if not through predefined interfaces, such as the inputs and outputs of the system. Afterwards, those cryptosystems are implemented into physical circuits, with the aim to carry out their specific tasks. This opens up the possibility for attackers to use so-called *side channels*, obtaining additional information about the internal computation of the systems, leaked by the circuit. Attacks that exploit these information are called Side Channel Attacks. A specific type of Side Channel Attacks (SCA) are Power Analysis Attacks (PAA); mounting a PAA, the attacker can obtain details about the internal calculations performed by the circuit analyzing its power consumption. Then, these information can be used to retrieve the secret key.
The final achievement of this thesis is to generate a tool which can analyse the weakness of a circuit towards PAA, as realistically as possible. We developed a tool, called VoLPE (Verification of Leakages Propagation Escalation) which simulates a synthesized circuit, obtained from OpenLane, and calculates the existing correlation between the inputs and a model describing the power consumption of the circuit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cryptography is the science that studies cryptosystems, whose goal is to protect communications from the intervention of undesired adversaries. This is done by encrypting the messages m of a communication with a secret key k, making m unintelligible to anyone that is not in possession of a decryption key $\tilde{k}$. Using the key the encryption of the message can be reversed, making m readable again. Any cryptographic algorithm is considered to be secure if it can not be broken in polynomial time, or in other words, if is not possible to efficiently recover the initial message or the decryption key. Indeed the security of any cryptographic algorithm should hold even if the algorithm is publicly known, basing therefore its security only on the secrecy of the used key (Auguste Kerckhoff, 1883).
Cryptanalysis, on the other hand, is the process of analysing cryptosystems with the goal of breaching them, trying to gain access to the content of the encrypted message without knowing the secret key.

For these cryptosystems to be used, they have to be implemented onto physical devices called circuits. While the security of the algorithms contained in these cryptosystems is well established, the security of their physical implementation is not. An attacker with physical access to the device can obtain information about the secret key and the internal state of the computation, by performing measurements on the power consumption and electromagnetic radiations of the device. These measurements can be performed by placing metal needles, called *probes*, on internal wires of the circuit. Attacks leveraging these measurements are called *Side Channel Attacks* (SCA). Side Channel Attacks that exploit the information leaked by the power consumption are called *Power Analysis Attacks*. Often these leakages are caused by the presence of *glitches* during the computations of the algorithms, which in turn are caused by the different propagation times of signals in the circuits.

A very powerful approach to protect physical circuits against SCAs are *masking schemes*, which aim at making the computations independent from the sensitive data they are processing. This is done by xoring the inputs of the circuit with a random number. Additionally a technique called *threshold implementations* can be implemented to further secure the sensitive variables. Threshold implementations do that by dividing each variable v, used during computations, in $n+1$ random and independent shares, whose exclusive or is equal to v. Gates performing non-linear operations are usually more vulnerable towards SCA. Vulnerable gates have to be replaced by a cluster of gates called *gadgets*, which perform the same operation of the gate they are replacing, using however the aforementioned masking schemes and threshold implementations.

In order to test the actual security of these countermeasures, models are created with the aim to capture the leaked information with similar capabilities of an attacker. In particular these models specify the information that can be captured with every measurement taken by the attacker. These attack models, however, cannot reproduce the actual physical implementation of the circuit and therefore insert approximations into the verification process.

1.1 Objectives

The aim of this work is to develop a tool able to determine the weakness of circuits towards Power Analysis Attacks as realistically as possible. Our tool does that by simulating a synthesized circuit, using *Icarus Verilog*, and calculating the existing correlation between the inputs and a model describing the power consumed during the computation. Starting from a high level description of the circuit, its synthesis is obtained using *OpenLane* and represents the actual implementation of the circuit on the chip. The power consumption is modeled with the number of *toggles* performed by the output of the circuit during the computations. Analysing the simulation of the actual implementation of circuits, we aim at overcoming the approximations introduced by the previously mentioned attack models.

1.2 Outline of contents

Chapter 2

In Chapter 2 we start by providing the mathematical background necessary to understand the rest of this thesis. Thereafter we give a description of the two cryptographic algorithms Xoodyak and AES, used later on to test our tool. We then proceed by introducing Side Channel Attacks, with particular attention on Power Analysis Attacks. After describing what physical defaults are and explaining how these leak information, we continue by discussing potential countermeasures. We conclude this chapter by giving some definitions used to identify the security level of circuits.

Chapter 3

A brief explanation of third party tools used during the implementation of VoLPE is given in Chapter 3. We continue by giving a step by step guide of the usage of OpenLane, which is necessary for the usage of our tool.

Chapter 4

Chapter 4 gives an in depth description of every component of the developed tool. We start by explaining the mandatory fields of the configuration file, proceeding then to analyse how VoLPE executes the simulation of the circuit. This chapter continues by giving an outline of the power consumption model and the consume model, before concluding with the description of the correlation function used to calculate the correlation matrix for the tested circuit.

Chapter 5

We conclude this work in Chapter 5 by discussing the results obtained with the tool on several implementations of four circuits. In particular we start with the tests performed on the AES S-Box, continuing with the tests performed on Xoodoo and finishing with results obtained with CMS and DOM.

Chapter 2

State of the art

2.1 Mathematical context

As everything in computer science, also cryptography works mainly with bits and functions transforming this binary data. Therefore this work will start by providing some mathematical background about the Binary Field $\mathbb{F}_2$ and Boolean Functions. Also other concepts like Hamming weight, Hamming distance and correlations are introduced in this first Section.

2.1.1 Binary Field

Definition 2.1 (**Field**). *A **field** is a set $\mathbb{F}$ with two composition laws $+$ and $\cdot$ such that [27] [6]*

1. *$(\mathbb{F},+)$ is a commutative group;*
2. *$(\mathbb{F}^*,\cdot)$, where $\mathbb{F}^* = \mathbb{F} \setminus 0$, is a commutative group;*
3. *the distributive law holds.*

The field, whose set $\mathbb{F}_2 = \{0,1\}$ contains only two elements, is called *binary field* and its elements *bits*.
For binary fields the aforementioned composition laws are called *and* ($\wedge$) and *xor* ($\oplus$) and are defined in Tables 2.1 and 2.2 respectively. These definitions are also called *truth tables*, showing the correspondence between Binary Fields and Boolean Logic.
In order to simplify some future definitions two additional operations are introduced, even though they could be expressed as combinations of $\oplus$ and $\wedge$.
Firstly the *not* ($\neg$ or $\overline{x}$, where $\overline{x} = \neg x$) operation that returns the opposite of a given element in $\mathbb{F}_2$ (Table 2.5). For simplicity we call the negation of the and operation *nand* and the negation of the xor operation *xnor*.
Secondly the *or* ($\vee$) operation whose truth table is defined in Table 2.6.

$\wedge$	0	1
0	0	0
1	0	1

Table 2.1: *and* operation in $\mathbb{F}_2$.

$\oplus$	0	1
0	0	1
1	1	0

Table 2.2: *xor* operation in $\mathbb{F}_2$.

$\overline{\wedge}$	0	1
0	1	1
1	1	0

Table 2.3: *nand* operation in $\mathbb{F}_2$.

$\overline{\oplus}$	0	1
0	1	0
1	0	1

Table 2.4: *xnor* operation in $\mathbb{F}_2$.

2.1.2 Boolean Functions

A *Boolean function* maps a vector $\mathbf{x} = [x_0, ..., x_n]$ of length n with $x_0, ..., x_n \in [0,1]$ to a single binary value [30]:

$$f : \{0,1\}^n \to \{0,1\} \tag{2.1}$$

Where the domain $\{0,1\}^n$ of the Boolean function is called *binary Hamming space* of dimension n (also called *binary n-cube*) [7].
Since the co-domain of the Boolean functions is $\mathbb{F}_2$, it is possible to to apply the operations $\wedge$, $\oplus$, $\neg$ and $\vee$ to their outputs.

2.1.3 Hamming weight and distance

Two further functions over vectors of $\mathbb{F}_2^n$ are relevant for the calculations discussed in Section 4.1.3, namely Hamming weight and distance.

Definition 2.2 (**Hamming distance**). *Given two vectors* $\mathbf{x} = [x_0, ..., x_n]$ *and* $\mathbf{y} = [y_0, ..., y_n]$, *the* ***Hamming distance*** *between them is defined as the number of indexes* i, *where* x_i *and* y_i *differ [30].*

$$d_h(\mathbf{x}, \mathbf{y}) = |\{i : x_i \neq y_i\}| \tag{2.2}$$

Definition 2.3 (**Hamming weight**). *Given a vector* $\mathbf{x} = [x_0, ..., x_n]$ *its* ***Hamming weight*** *is defined as*

$$w_h(\mathbf{x}) = d_h(\mathbf{x}, \mathbf{0}) \tag{2.3}$$

where $\mathbf{0}$ *is the all-zero vector* $\mathbf{0} = [0, ...,0]$ *[7].*

In other words, the Hamming weight represents the number of bits of $\mathbf{x}$ that are set to 1.

$\neg$	
0	1
1	0

Table 2.5: *not* operation in $\mathbb{F}_2$.

$\vee$	0	1
0	0	1
1	1	1

Table 2.6: *or* operation in $\mathbb{F}_2$.

Example 2.1. *Given two vectors* $\mathbf{x} = [\mathbf{1}, \mathbf{0}, 0, \mathbf{1}, 1, 1]$ *and* $\mathbf{y} = [\mathbf{0}, \mathbf{1}, 0, \mathbf{0}, 1, 1]$ *(in bold the elements that differ in the two vectors)*

$$d_h(\mathbf{x}, \mathbf{y}) = 3 \tag{2.4}$$

$$w_h(\mathbf{x}) = 4 \tag{2.5}$$

2.1.4 Correlations

Correlation is a statistical measure used to quantify the linear relation between two variables, where the correlation coefficient defines the strength of this relation [4]. Given two variables x and y, which assume the values $\{x_0, ..., x_n\}$ and $\{y_0, ..., y_n\}$, respectively, their so-called *Pearson correlation coefficient* (or *product moment correlation coefficient*) is given by the following equation:

$$r = \frac{\sum_{i=1}^{n}(x_i - \mathbb{E}(x))(y_i - \mathbb{E}(y))}{\sqrt{\sum_{i=1}^{n}(x_i - \mathbb{E}(x))^2 \sum_{i=1}^{n}(y_i - \mathbb{E}(y))^2}} \tag{2.6}$$

$\mathbb{E}(x)$ and $\mathbb{E}(y)$ are the mean values of x and y. The Pearson correlation coefficient can assume values between 1 and -1. A value close to 1 indicates a strong positive linear relation between the two considered variables. A value close to -1, on the other hand, stands for a negative linear relationship. A value equal to 0, on the contrary, indicates the absence of a linear relation between x and y.

2.2 Cryptography

Cryptography deals with the creation of cryptosystems, which are used to secure the communication between two or more entities against an untrusted third party [21]. A cryptosystem can be defined by a tuple $(\mathcal{P}, \mathcal{C}, \mathcal{K})$ with the following definitions:

- $\mathcal{P}$ is the plaintext space. Its elements are the messages that have to be sent in a secure way.
- $\mathcal{C}$ is the ciphertext space. It contains the encrypted plaintexts.
- $\mathcal{K}$ is the key space. This finite set contains the keys that are used for encryption.

and the three algorithms **Gen**, **Enc** and **Dec** [19]:

- **Gen** is the algorithm generating the keys $k \in \mathcal{K}$.

- **Enc** is the encryption algorithm which takes as inputs a key $k \in \mathcal{K}$ and a plaintext $p \in \mathcal{P}$ and generates the ciphertext $c \in \mathcal{C}$.

- **Dec** is the decryption algorithm which takes as inputs a key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$ and outputs the plaintext $p \in \mathcal{P}$.

Based on the keys used during encryption and decryption, the cryptosystems can be divided into *symmetric-key* or *public-key* cryptosystems.

As the name suggests, in *symmetric-key cryptography* the same key is used both for the encryption of the plaintext and the decryption of the ciphertext and must therefore be shared among all entities involved in the communication.
Given an *encryption function* $\mathbf{Enc_k}$ and a *decryption function* $\mathbf{Dec_h}$ with $h, k \in \mathcal{K}$

$$\mathbf{Dec_h}(\mathbf{Enc_k}(p)) = p \quad \forall p \in \mathcal{P} \; and \; h = k \tag{2.7}$$

Two different types of algorithms can be used in symmetric cryptography:

- *Stream ciphers:* In order to encrypt a plaintext using a stream cipher, the former has to be xored bitwise with a stream of pseudorandom bits, called *keystream*. The keystream is generated starting from a short secret seed and has the same length as the plaintext that has to be encrypted. Given the plaintext p as a sequence of bits $p = p_0 p_1 ... p_{n-1}$ and the keystream k as $k = k_0 k_1 ... k_{n-1}$, then the ciphertext $c = c_0 c_1 ... c_{n-1}$ is defined as $c_i = p_i \oplus k_i$ for $i \in [0, n-1]$ [33]. Since the same key is also used for decryption, it follows that $p_i = c_i \oplus k_i$ for $i \in [0, n-1]$.

- *Block ciphers:* As opposed to stream ciphers, block ciphers don't work on the entire plaintext at once, but rather on fixed sized parts, called *blocks* $(b_0, ..., b_{n-1})$ [21]. For the encryption of each block r rounds are needed. In order to have a different key for each round, a set of so-called *round keys* $(k_0, ..., k_{r-1})$ is generated by an algorithm called *key schedule*, starting from the key k. Let r_i be the intermediate result of round i then:

$$r_i = \begin{cases} \mathbf{Enc_{k_0}}(b_0) & \text{if } i = 0 \\ \mathbf{Enc_{k_i}}(r_{i-1}) & \text{if } i \in [1, r-1] \end{cases} \tag{2.8}$$

  The result of the last round is the ciphertext $c = r_{r-1}$. This steps must be repeated for each block composing the original plaintext.

In contrast to the previous case, in *public-key cryptography* different keys are used for encryption and decryption, called *public* (k_pub) and *private key* (k_priv). Given an encryption function E_k and a decryption function D_h with $h, k \in \mathcal{K}$

$$\mathbf{Dec_h}(\mathbf{Enc_k}(p)) = p \quad \forall p \in \mathcal{P} \; and \; h \neq k \tag{2.9}$$

Public-key cryptography can be used in two modes [34]:

- If k_pub is used to encrypt the plaintext, then only the entity owning the corresponding k_priv can decrypt the ciphertext, guaranteeing the confidentiality of the message. Asymmetric cryptography being way slower than its symmetric counterpart, is usually used in this way, to distribute a symmetric key. Symmetric cryptography is then used for the rest of the communication.

- If k_priv is used to encrypt the plaintext, everybody can use k_pub to decrypt the message. When used in this way the origin of the message is guaranteed, since only the owner of k_priv can be the sender of the message, ensuring its authenticity.

2.2.1 Advanced Encryption Standard

In response to the growing concerns about the security of the *Data Encryption Standard* (DES), the National Institute of Standards and Technology (NIST) organized an open process lasting from 1997 to 2000, in order to find a new encryption standard to substitute DES. A restricted version of the symmetric block cipher, named *Rijndael*, submitted by the two Belgian cryptographers Joan Daemen and Vincent Rijmen, was finally selected as *Advanced Encryption Standard* (AES) in 2001.

AES encrypts and decrypts input blocks of size 128 bits using keys of size 128-bit, 192-bit or 256-bit length [17] and returns output blocks of 128 bits [13]. During its execution the cipher operates on a 128-bit long block, called *state*, internally organized as a 4x4 table of bytes:

$$\mathbf{S} = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \tag{2.10}$$

A byte is a vector of 8 bits: $\mathbf{B} = [b_7, ..., b_0]$, which can be interpreted as element of the finite field $\mathbb{F}_2^8$ using its polynomial representation:

$$\mathbf{B}(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^{7} b_ix^i \tag{2.11}$$

In this representation, the coefficients of the polynomial are equal to the value of the corresponding bit. A given byte **B** = [0,1,1,0,0,0,1,0] is represented by the following polynomial:

$$\mathbf{B}(x) = x^6 + x^5 + x \tag{2.12}$$

The sum of two bytes (denoted by $\oplus_8$) consists in the sum of the single bits, composing those bytes taking into consideration the carry over from the previous bit. Given two bytes $\mathbf{A} = [a_7, ..., a_0]$ and $\mathbf{B} = [b_7, ..., b_0]$, then the sum is $\mathbf{C} = [c_7, ..., c_0] = \mathbf{A} \oplus_8 \mathbf{B}$, where $c_i = (a_i \oplus b_i) \oplus r_{i-1} \forall i \in [0,7]$, where r_{i-1} is the carry over from the xor of the previous bits.
Whereas the multiplication in $\mathbb{F}_2^8$, corresponds with the multiplication of polynomials modulo an *irreducible polynomial* of degree 8. A polynomial is irreducible if it can be divided only by one and itself [13]. The irreducible polynomial chosen by AES is:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{2.13}$$

The set of bytes $\mathbb{F}_2^8$ together with the sum operation $\oplus_8$ and the polynomial multiplication form a field, as defined in Definition 2.1.

Before starting, the input block is copied into the state and the input key is added. Thereafter a *round function* is applied to the state Nr times (see Table 2.7 for the values of Nr), with the last round function slightly differing from the previous ones [31] [17].
In order to have a different key for each round AES includes a *Key Expansion* step generating a *key schedule*. The key schedule consists of $4 * (Nr + 1)$ bytes: 4 bytes for the initial `AddRoundKey` and other 4 bytes for the `AddRoundKey` of the following Nr rounds.
The round function consists of four steps:

1. `SubBytes`: This step uses a table called *S-Box* (or *substitution table*) on each byte of the state independently. The S-Box is invertible and corresponds to applying a non linear inversion in $\mathbb{F}_2^8$, followed by an affine transformation over $\mathbb{F}_2$ to the byte in input. Using the S-Box these steps are equivalent to substituting a byte with the corresponding byte in the S-Box (see Table 2.8). For example the byte $B = \{01010011\}_2 = \{53\}_{16}$ would be substituted with the value corresponding to the line with index '5' and the column with index '3'. The new value of **B** after `SubBytes` would be $\{ed\}_{16}$. This step represents the only non-linear function of the cipher [11].

2. `ShiftRows`: In this step every byte of each row is shifted cyclically to the left by an offset corresponding to $(l - 1)$ where l is the line number. Bytes in the first row are therefore not shifted at all, bytes in the second row are shifted by one position, in the third row they are shifted by two positions and in the last row bytes are shifted by 3 positions to the left [2] (see Figure 2.1).

3. `MixColumns`: Considering the columns of the state as polynomials over $\mathbb{F}_2^8$, they are multiplied by a fixed matrix:

$$\begin{bmatrix} s'_{i,0} \\ s'_{i,1} \\ s'_{i,2} \\ s'_{i,3} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} * \begin{bmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{bmatrix} \tag{2.14}$$

   This step is skipped in the last round, since it wouldn't add any additional security.

4. `AddRoundKey`: Using the $\oplus_8$ operation, described above, each byte of the state ($\mathbf{S}$) is added to the corresponding byte of the round key (rk):

$$s' = \{(s_7 \oplus rk_7)(s_6 \oplus rk_6)(s_5 \oplus rk_5)(s_4 \oplus rk_4)(s_3 \oplus rk_3)(s_2 \oplus rk_2)(s_1 \oplus rk_1)(s_0 \oplus rk_0)\}$$

   The round keys are extracted from the key schedule, generated during the Key Expansion.

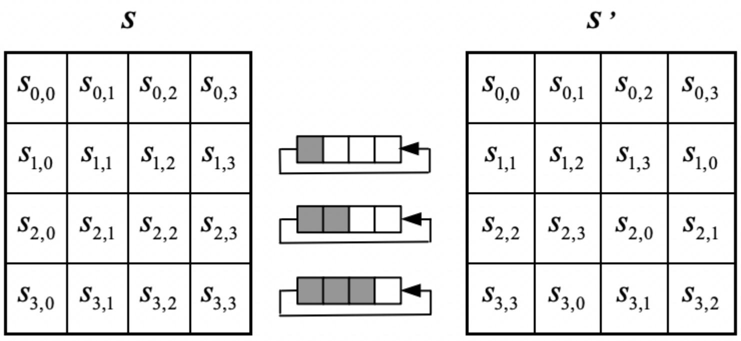
2.2.2 Xoodyak

Another cryptosystem of relevance for this work is *Xoodyak*, which internally builds upon the *Xoodoo* permutations.
As AES, *Xoodoo* applies a round function to an internal state for a total of n_r rounds. The number of rounds is a parameter of the algorithm. The chosen value for a specific implementation can be indicated as Xoodoo[n_r] [9].
The internal state consists of three horizontal *planes*, each of them composed of four *lanes* of 32 bits. The state can therefore be represented as an array containing three planes, each of which can be represented as a 4×32 matrix. Each plane is indexed by $y \in [0,2]$, where $y[0]$ indicates the lowest plane and $y[2]$ the highest one. The bits inside a lane are indexed with z, whereas the lane inside a plane is indexed by x. A *column* is composed by the the three bits in the same position on the three planes, which can be identified by the tuple (x, z). Given an internal state, the position of the lanes can be defined by the tuple (x, y), whereas a single bit is identified by (x, y, z). Three lanes positioned on top of each other are called *sheets*, and can be identified by x. The structure of the internal state can be seen in Figure 2.2.

The round function performed by Xoodoo is divided into five steps:

- θ: An initial mixing layer is applied to the internal state. The results of this layer can be seen in Figure 2.4.

- ρ_{west}: A first shifting function is applied on the top two planes of the state. The functioning of this permutation can be seen on the right side of Figure 2.5.



s
s'
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

Figure 2.1: Effect of `ShiftRows` on the state [11]

	Key Length (Kl)	Block Size (Bs)	Number of Rounds (Nr)
AES-128	128	128	10
AES-192	192	128	12
AES-256	256	128	14

Table 2.7: Key-Block-Round combinations AES [13]

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
x	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table 2.8: Substitution values for the byte xy in hexadecimal format

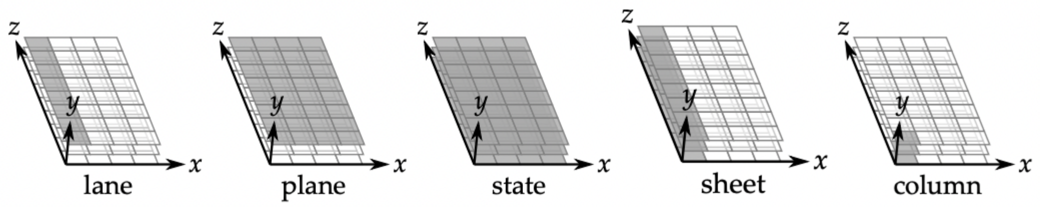


Figure 2.2: State representation of Xoodoo [10]

- ι: In this step a round constant is added to the first plane of the state. The round constants, which are added to the state in this step, are represented by planes with only one non-zero lane at $x = 0$. Their hexadecimal values can be seen in Table 2.9.
- χ: This step represents the only non linear step of the algorithm and can be seen in Figure 2.3.
- ρ_{east}: The last step of each round is another shift function, applied on the same planes as ρ_{west} (see left side of Figure 2.5).

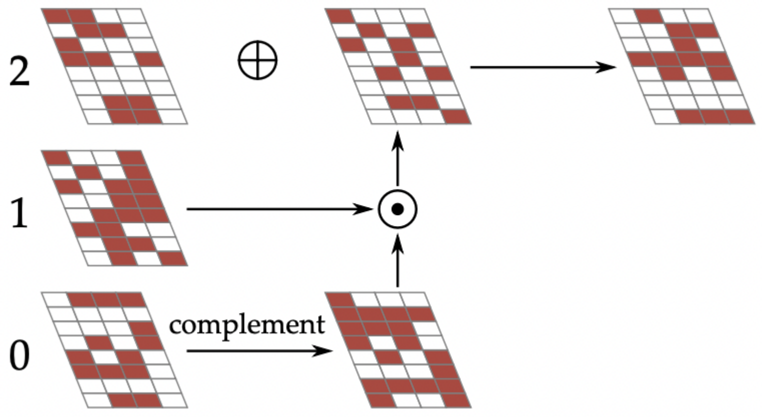


Figure 2.3: χ permutation of Xoodoo [10]

When the n_r rounds are finished, in many applications the internal state is converted into a 348-bit array s, where every element s_i is obtained with the following conversion formula:

$$s_i = z + 32(x + 4y) \tag{2.15}$$

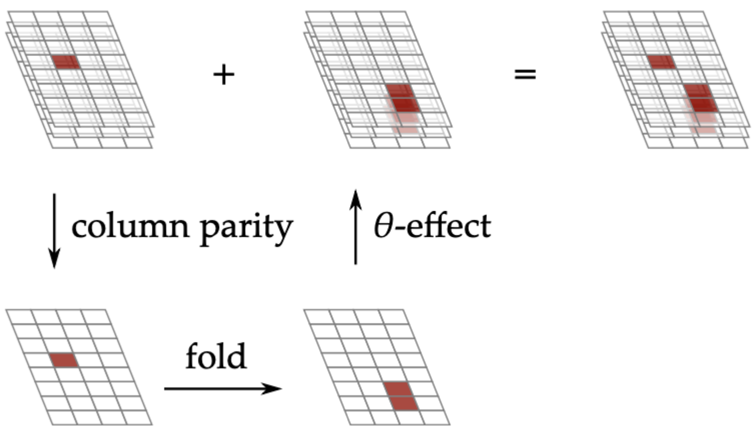


Figure 2.4: θ permutation of Xoodoo [10]

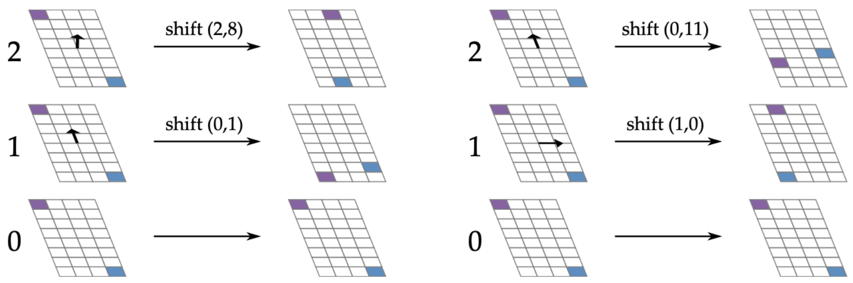


Figure 2.5: ρ_{east} (left) and ρ_{west} (right) permutations of Xoodoo [10]

2.3 Circuits

The implementation of Boolean functions and cryptographic algorithms is performed using Circuits. Circuits can be divided into two categories, depending on their ability to maintain an internal state [26] [22]:

- *Combinational:* A circuit can be defined as *combinational*, if its output depends only on the current inputs, applied to the circuit.
- *Sequential:* On the contrary a circuit is called sequential if its output is also dependent on an internal state, saved during some previous iterations.

Circuits can also be further divided into cyclic and acyclic, based on the presence or not of internal loops.

Any circuit is composed of a variable number of *logic gates*, implementing the basic operations of Boolean algebra (see Figure 2.1.2). Optionally circuits may also

Round	Round constant
0	0x00000058
1	0x00000038
2	0x000003C0
3	0x000000D0
4	0x00000120
5	0x00000014
6	0x00000060
7	0x0000002C
8	0x00000380
9	0x000000F0
10	0x000001A0
11	0x00000012

Table 2.9: Xoodoo round constants for $n_r = 12$

contain *registers*, which are used to temporarily save an internal state. Logic gates are interconnected amongst themselves and with registers through wires.

Circuits may or may not use a so-called *clock* for internal synchronization; a clock is a square wave signal oscillating between a low and a high state, with a *clock cycle*, defined as the interval between two subsequent rises of the signal (see Figure 2.6). By default, the rising edge of the clock is used as activation signal for registers, causing them to read the data on their input and transmitting their internal value to the output.

A circuit, using a clock for the synchronization of its internal operations, is trivially called *synchronous circuit*, as opposed to an *asynchronous circuit*, which performs all operations asynchronously.

While ideally the circuit immediately changes its output once the inputs change, in reality every component will introduce a so-called *delay*. These delays are due to the non-zero propagation time of signals through the physical components (see figure 2.7).

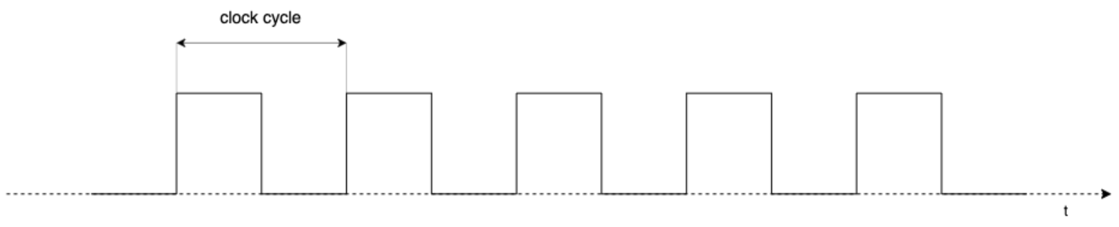


Figure 2.6: Clock signal with clock cycle

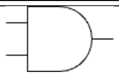
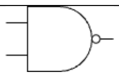
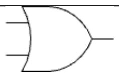
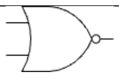
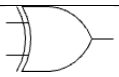
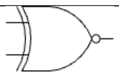
Name	Logic gate	Boolean operation
and		$and(x, y) = x \wedge y$
nand		$nand(x, y) = \neg and(x, y)$
or		$or(x, y) = x \vee y$
nor		$nor(x, y) = \neg or(x, y)$
xor		$xor(x, y) = x \oplus y$
xnor		$xnor(x, y) = \neg xnor(x, y)$

Table 2.10: Logic gates and corresponding Boolean operations

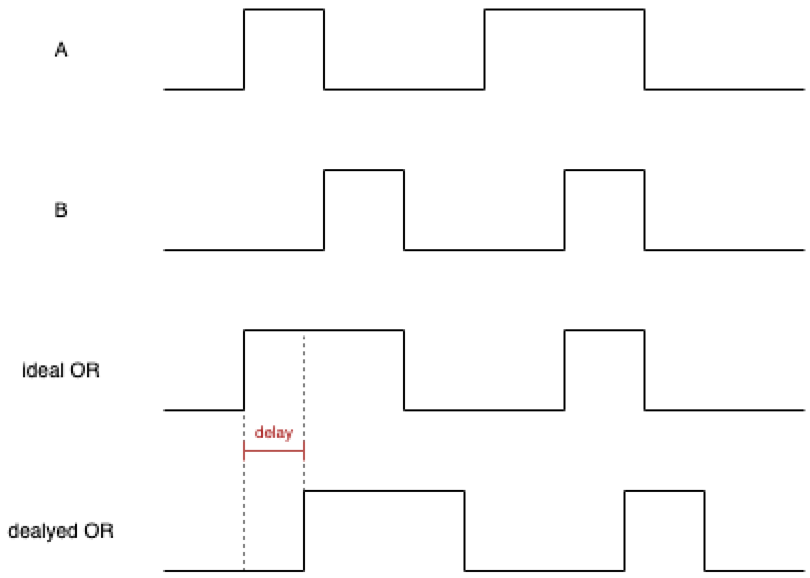


Figure 2.7: Propagation delay

2.4 Side Channel Attacks

The security of cryptosystems is usually evaluated considering them as *black boxes*; the adversary can access the cryptosystem only through predefined interfaces, i.e. the inputs and outputs of the system [8]. In reality, cryptosystems have to be implemented onto physical devices, which will leak some additional information (*leaks*) about the internal computations, giving the adversary access to intermediate values.

Definition 2.4 (**Side Channel**). *The measurements, giving the adversary access to leaks, are called* ***Side Channels****.*

Definition 2.5 (**Side Channel Attacks (SCA)**). *Attacks performed, using the information obtained through side channels are called* ***Side Channel Attacks****.*

Some possible side channels are the execution time of the algorithm, the power consumed during the execution and electromagnetic fields, produced by the device. In this work only *Power Analysis Attacks*, which exploit leaks obtained from the power consumption of the devices, are discussed (see Section 2.4.1).

Side channels attacks can be classified as follows [23]:

- *Invasive/non-invasive:* In order to perform an invasive SCA, the adversary has to gain access to the physical device on which the cryptosystem is implemented, and operate on it. An example could be the insertion of a probe on a wire to see the transferred data. On the contrary, a non-invasive SCA uses information available externally to the device, such as execution time.

- *Active/passive:* The objective of active SCAs is to tamper the usual functions of the attacked device. An attacker could, for instance, try to inject errors into the computation. On the other hand, passive attacks settle for observing the normal execution, trying to extract useful information.

2.4.1 Power Analysis Attacks

Power Analysis Attacks (PAA) are a powerful type of side channel attacks, that can be performed quite easily, making them particularly interesting for further research. Each operation in a physical device consumes a certain amount of power, which, as result, depends on that specific operation. Then, analysing the power consumption, one can obtain details about the internal calculations of the device.

In order to perform a PAA, the attacker has to have access to the device and to be able to control its execution [25]. In this way the attacker is able to map each performed operation with the corresponding power consumption.

Definition 2.6 (**Power traces**). *The collected measurements about the power consumption of a device are called* ***power traces****. A power trace $V \in \mathbb{R}^T$, with $T \in \mathbb{N}$, contains the power consumption at each considered point in time $t \in T$.*

After an initial step, where sufficient power traces are collected, various types of power analysis attacks can be performed, differing in the method used to analyse the obtained traces:

- *Simple Power Analysis (SPA):* Using SPA it is possible to recover the executed operations and the order of execution [20]. This analysis method is typically used as initial information gathering step, before mounting a more complex attack. In order to fully exploit the obtained traces, it is useful for the attacker to have detailed information about the algorithm implementation, and the architecture of the device it is implemented on.

- *Differential Power Analysis (DPA):* DPA uses statistical analysis, to gather side channel information. A DPA attack on a given encryption algorithm A is performed in several steps:

  1. Given a set of n randomly generated plaintexts $P = \{p_0, ..., p_{n-1}\}$, the algorithm A is executed n times. For each execution the power trace V_{p_i}, generated by plaintext p_i, is saved. The resulting ciphertext can be saved as well, if needed.
  2. For the next step the attacker has to choose a key-dependent selection function $S = (P, K')$. K' is a guess of the key, made by the attacker. This function is then used to partition the power traces, obtained in the previous step, into two subsets [25]:

     $$S_0 = \{V_{p_i} | S(p_i, K') = 0\} \tag{2.16}$$
     $$S_1 = \{V_{p_i} | S(p_i, K') = 1\} \tag{2.17}$$

  3. The average for both sets is calculated as:

     $$\mathbb{E}(S_i) = \frac{1}{|S_i|} \sum_{V_{p_i} \in S_i} V_{p_i} \forall i \in \mathbb{Z}_2 \tag{2.18}$$

  4. The difference of the two obtained means $M = \mathbb{E}(S_0) - \mathbb{E}(S_1)$ is computed. If the used selection function divides the traces randomly in the two sets S_0 and S_1, then the value of M should approach zero for an increasing number of traces, indicating a missing correlation between the chosen selection function and the actual calculations performed by the observed device. Choosing a correct K' the selection function will be able to predict the correct subset for each trace, with a higher probability, resulting in a non-zero value for M [20].

- *Correlation Power Analysis (CPA):* CPA attacks are performed in the following steps:

  1. Starting with a set of n known data $D = \{d_0, ..., d_n\}$, for each data the corresponding power trace t_i is calculated. If each trace is considered as a vector containing m different samples ($\mathbf{t}_i = [t_{i,0}, ..., t_{i,m-1]}$), we obtain

the following matrix:

$$T = \begin{bmatrix} \mathbf{t}_0 \\ \mathbf{t}_1 \\ \vdots \\ \mathbf{t}_{n-1} \end{bmatrix} = \begin{bmatrix} t_{0,0} & \dots & t_{0,m-1} \\ t_{1,0} & \dots & t_{1,m-1} \\ \vdots & \ddots & \vdots \\ t_{n-1,0} & \dots & t_{n-1,m-1} \end{bmatrix} \tag{2.19}$$

2. The divide and conquer paradigm is used to guess the key, working only on small portions, referred to as *subkeys*. For each known data $d_i \in D$ and each possible value of the subkey s_j, an intermediate value $f(d_i, s_j)$ of the algorithm is calculated, resulting in the following matrix:

$$V = \begin{bmatrix} f(d_0, s_0) & \dots & f(d_0, s_{x-1}) \\ f(d_1, s_0) & \dots & f(d_1, s_{x-1}) \\ \vdots & \ddots & \vdots \\ f(d_n - 1, s_0) & \dots & f(d_{n-1}, s_{x-1}) \end{bmatrix} = \begin{bmatrix} v_{0,0} & \dots & v_{0,x-1} \\ v_{1,0} & \dots & v_{1,x-1} \\ \vdots & \ddots & \vdots \\ v_{n-1,0} & \dots & v_{n-1,x-1} \end{bmatrix} \tag{2.20}$$

3. After finding an adequate power model $f_{pm}(v_{i,j})$, it is applied to the intermediate values, returning a hypothetical power consumption for each value, resulting in the following matrix:

$$P = \begin{bmatrix} f_{pm}(v_{0,0}) & \dots & f_{pm}(v_{0,x-1}) \\ f_{pm}(v_{1,0}) & \dots & f_{pm}(v_{1,x-1}) \\ \vdots & \ddots & \vdots \\ f_{pm}(v_{n-1,0}) & \dots & f_{pm}(v_{n-1,x-1}) \end{bmatrix} \tag{2.21}$$

   Two possible power models are $w_h(v_{i,j})$ or $d_h(d_i, v_{i,j})$ (see 2.1.3).

4. In order to find the subkey s_j that is most likely to be correct, the pearson's correlation (see Section 2.1.4) between each column T_i of T and each column P_i of P is computed. The column with the highest correlation coefficient probably corresponds to the correct value of s_j.

2.4.2 Physical Defaults

As discussed in Section 2.4, SCAs work with information, gathered by performing measurements on the physical devices, cryptographic algorithms are computed on. Some physical defaults of those devices may, however, even increase the amount of information that can be gained. Physical defaults can be divided into one of these types:

- *Combinatorial recombinations (glitches):* These physical defaults are caused by the intrinsic propagation delays, present in physical circuits; indeed such

delays may cause an input A of a gate to arrive before the input B of the same gate. As a consequence of this delay, the output of the gate will change as a reaction to input A and then again reacting to input B. This unexpected temporary change of the output, between the arrival of the two inputs introduces a brief time span, during which the output may assume a wrong value [24].
To generate a glitch, the delay input B has with respect to input A, needs to be greater, than the rise time of the gate. In other words, the result generated by the arrival of input A has to propagate to the output of the gate before the arrival of input B. The first two waves of Figure 2.8 show how input A and an input B change their value at different points in time. The third wave shows the case, in which the raise time of the XOR gate is longer than the delay between A and B. As can be seen from the red dotted line, the output of the gate starts raising as soon as input A assumes a new value. However before the output can finish changing its value the change of B occurs, pulling the output down again. As a result no variation of the output can be observed. The last wave shows how the output of a XOR gate with shorter raise time assumes briefly a a wrong value.
In order to mitigate the effects of glitches, registers can be placed throughout the circuit; given that a register releases all the outputs at the same time, the delays of those signals are reset to zero.

- *Memory recombinations (transitions):* Memory recombinations mix and recombine the content of memory elements (i.e. registers), when invoked in two consecutive clock cycles. This happens if an old value x is erased from the memory element and a new value y is saved in the same cycle [14].

- *Routing recombinations (couplings):* Couplings may mix the data transferred over two adjacent wires [29].

2.4.3 Countermeasures

With the improvement of Side Channel Attacks and an increase in their popularity, new countermeasures have been created and old ones have been themselves improved. The two main countermeasures covered in this work are *masking* and *threshold implementations.*

Masking

With the introduction of a random *mask*, this countermeasure tries to randomize the power consumption, making it independent from the intermediate values of the computations. In order to do so a random value m is either added or multiplied to

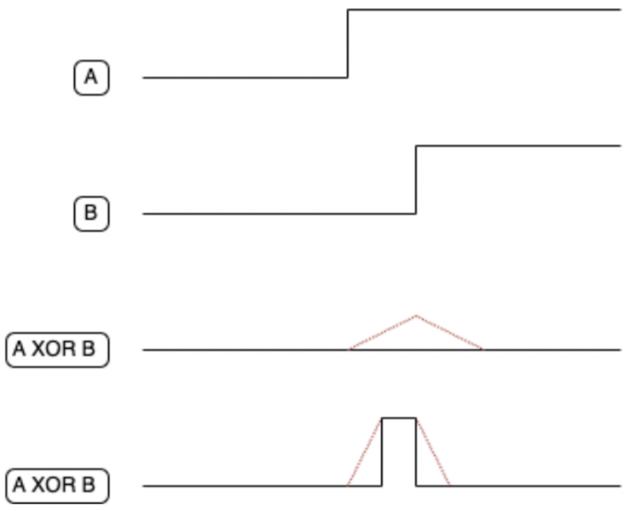


Figure 2.8: Condition for the presence of Glitches.

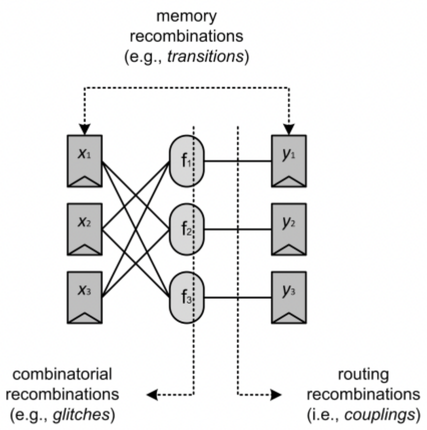

memory recombinations (e.g., transitions)
x1
x2
x3
f1
f2
f3
y1
y2
y3
combinatorial recombinations (e.g., glitches)
routing recombinations (i.e., couplings)

Figure 2.9: Possible physical defaults [14]

each intermediate value a of the circuit. The first, and more widespread scheme, is called *additive masking*, where the masked value a_m is obtained by *xoring* a and m: $a_m = a \oplus m$. While the second one is called *multiplicative masking*. Here intermediate value and mask are combined using a multiplication: $a_m = a * m$.
To guarantee the correct functioning of both schemes, the mask has to be added before starting any computation on the data and removed only when all computations are done. In order tho achieve the correct final result, while working with modified data, the implementation of some operations in the algorithm have to be changed.

Threshold implementations

Instead, threshold implementations divides the variables used for computations into $n+1$ uniformly distributed variables, called shares [5]. Therefore the original variable $x \in \mathbb{Z}_2$ is obtained by the addition in $\mathbb{Z}_2$ of the shares s_i, where $i = 0, ..., n$.

$$x = \bigoplus_{i=0}^{n} s_i \tag{2.22}$$

The first n shares are generated randomly, while the last one is computed such that Function 2.22 holds. Therefore the knowledge of n shares of x do not give any information about the starting variable x.

Given a vector $\mathbf{s} = [s_0, ..., s_n]$ of shares of a variable x, then the set $S(x)$ contains all possible valid shares of x:

$$S(x) = \{\mathbf{s} | s_0 \oplus s_1 \oplus ... \oplus s_n = x\} \tag{2.23}$$

Whereas $P(\mathbf{S} = \mathbf{s} | X = x)$ defines the probability of a certain vector of shares $\mathbf{S}$ being equal to $\mathbf{s}$, given that the un-shared variable X is equal to x. Given that f(X)=Y in threshold implementations is a function from $\mathbb{Z}_2^n$ to $\mathbb{Z}_2^m$, it has to be implemented as a vector of functions $\mathbf{f} = [f_0, ..., f_m]$. Every function contained in the vector is called *component function*.

In order to protect against Side Channel Attacks, the implementation must satisfy the following properties [29]:

Property 1 (**Correctness**). *Given* $\mathbf{x} = [x_0, ..., x_n]$ *and* $\mathbf{y} = [y_0, ..., y_m]$ *then* $\mathbf{y} = \mathbf{f}(\mathbf{x})$ *is correct iff* $y = \sum_i f_i(\mathbf{x})$ *for each* $\mathbf{x} \in S(x)$.

Property 2 (**Uniformity**). *A masking is uniform iff for all x there exists a constant p such that if* $\mathbf{x} \in S(x)$

$$P(\mathbf{X} = \mathbf{x} | X = x) = p \tag{2.24}$$

Property 3 (**Non-Completeness**). *For a masking to be non-complete every component function f_i of* $\mathbf{f}$ *has to be independent of at least one input share.*

If these three properties are respected, each output share y_i is independent of each input variable, and each output variable making the implementation secure against side channel attacks.

DOM

As first example of a masking scheme, in the following section an overview of the *Domain-Oriented masking* (DOM) is given.
The basic idea behind DOM is the introduction of so-called *share-domains*; in

domain-oriented masking each share of a variable is associated to a different domain [16]. This is also reflected by the different notation used to identify the shares. Given for example two variables x and y, their shares will be identified as A_x, B_x and A_y, B_y. Therefore the two variables can be retrieved as $x = A_x + B_x$ and $y = A_y + B_y$. In this notation A and B identify the two different domains. In order to achieve a d-probing secure gadget, $d + 1$ shares have to be used, which imply $d + 1$ different domains.
As long as the different domains can be kept separate during the computations their independence is guaranteed. This is not possible during non linear operations, where different domains necessarily have to be combined. To prevent the dependence of the shares even during non linear operations, DOM adds a fresh random share and a register when domain borders are crossed. The latter prevents generated glitches to propagate beyond this point.

Definition 2.7 **(n-DOM).** *A DOM gadget is identified as n-DOM, if the two variables, with which the calculations have to be performed, are divided into n domains each. In other words n-DOM is a $(n-1)$-probing secure DOM gadget.*

A 2-DOM multiplier gadget performs 3 steps to calculate the two shares of the result q, where $q = x * y$:

1. *Calculation*: Given the two input variables $x = A_x + B_x$ and $y = A_y + B_y$, this step calculates the actual multiplication $x * y = (A_x + B_x) * (A_y + B_y)$, calculating the product terms (A_xA_y), (B_xB_y), (A_xB_y) and (B_xA_y). Here (A_xA_y) and (B_xB_y) are called *inner-domain* terms, while (A_xB_y) and (B_xA_y) are called *cross-domain* terms. While inner-domain terms don't present any critical issues for the security, cross-domain terms require the variables x and y to be independently shared. If the independence can not be guaranteed, calculating (A_xB_x) would trivially leak information about the variable x, since $x = A_xB_x$. On the other hand combining two shares of different variables from different domains does not adversely affect the d^{th}-order security of the gadget. Circuit parts performing cross-domain operations are shown in red in Figure 2.10.

2. *Resharing*: During this step the cross-domain terms need to be modified, in order to allow their insertion in into an arbitrary domain. This is achieved by adding a fresh random share Z_0 to each of them. In order to prevent glitch propagation to the next step a final register is inserted. The registers shown with a dotted grey line in Figure 2.10, can be added to align inner-domain with cross-domain terms.

3. *Integration*: In this concluding step cross-domain and inner-domain terms are added resulting in $A_q = (A_xA_y) + [(A_xB_y) + Z_0]$ and $B_q = (B_xB_y) + [(B_xA_y) + Z_0]$.

The 2-DOM gadget can easily be expanded to a n-DOM gadget, considering that for two cross-domain terms, dependent on a common share, a different fresh random share has to be used. The same random share can for example be added to (A_xB_y) and (A_yB_x), but not to (A_xB_y) and (A_xC_y), since both terms depend from A_x. $(n-1)n/2$ fresh random shares are needed for a n-DOM gadget, as can be seen in Figure 2.11.

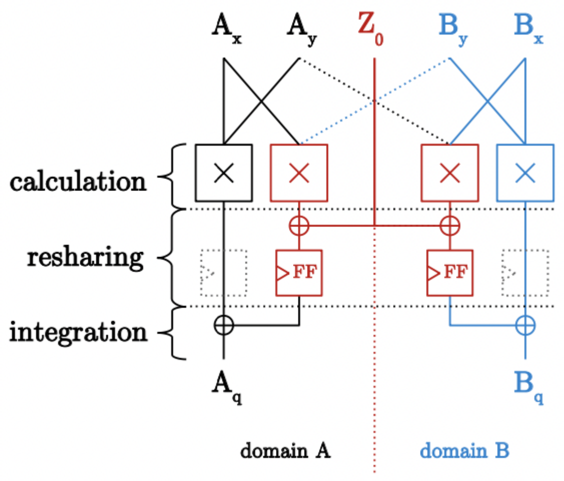


Figure 2.10: Structure of a 2-DOM [16]

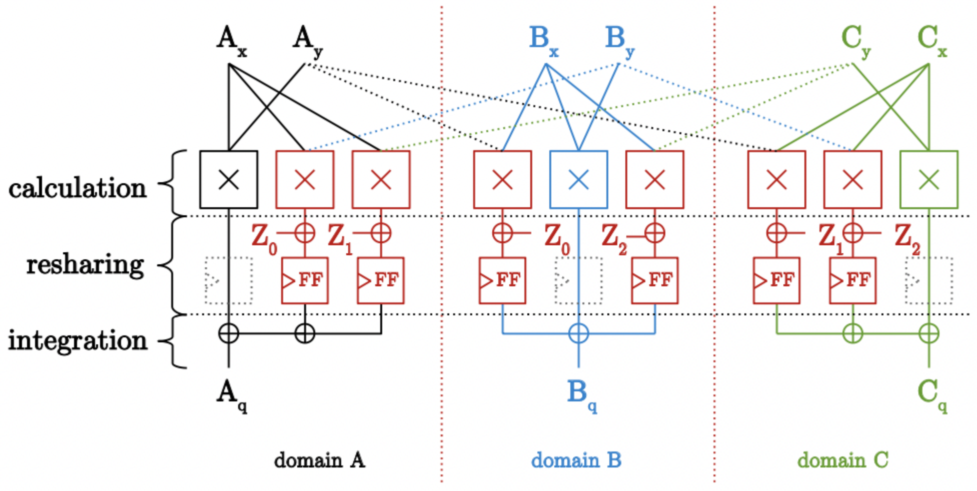


Figure 2.11: Structure of a 3-DOM [16]

CMS

A second example of masking scheme is given by the *Consolidating Masking Scheme* (CMS) proposed in [32]. A multiplication gadget constructed with the CMS structure can be divided into several layers:

- *Non-linear layer* $\mathcal{N}$: This layer performs the main computation, calculating all the products of the two variables, that need to be multiplied. Given for example two input variables, each divided into three shares, $a = a_1 + a_2 + a_3$ and $b = b_1 + b_2 + a_3$, $\mathcal{N}$ produces all product terms $a_i b_j$, generating 9 terms $(a_1 b_1, a_1 b_2, ..., a_3 b_3)$.
- *Linear layer* $\mathcal{L}$: Working with the amount of shares generated in the previous layer would become unpractical in higher order gadgets. For this reason this layer is introduced, which reduces the number of shares for the following steps, preserving the non-completeness property [12]. In a gadget working with three shares per variable the number of intermediate terms is reduced from 9 to 3.
- *Refreshing layer* $\mathcal{R}$: In order to remove dependencies between the results, generated in the previous layer, fresh random shares are added in $\mathcal{R}$.
- *Synchronization layer* $\mathcal{S}$: As for DOM a final layer of registers is inserted in this layer, in order to prevent glitch propagation to the next layer and output.
- *Compression layer* $\mathcal{C}$: To further reduce the number of output shares produced by the gadget, the final compression layer can be used. This layer is useful for the generation of multiplication gadgets with the desired number of output shares.

In Figure 2.12 two sample CMS gadgets can be seen, using 5 shares in the left example and 3 shares on the right.

2.4.4 Probing security

After having discussed Power Analysis Attacks in Section 2.4.1, in this section *Probing Attacks* are introduced.
In order to analyze the security of masked algorithm implementations the *d-probing model* was introduced by [18]. In this model an attacker has the capability of placing t *probes* on the wires of the attacked circuit. Through these probes different measurements can be performed, revealing for example the value carried on the wire. The number of probes, the attacker is able to place on the circuit, specifies the *order* t of the attack. In order to protect a circuit from a t-order Probing Attack every sensitive value x must be divided into $t + 1$ shares.

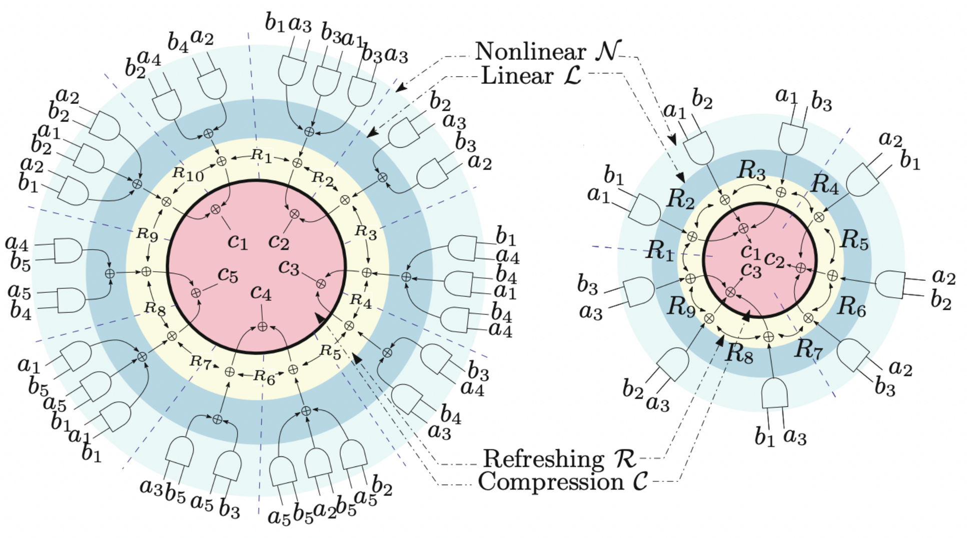


Figure 2.12: Structure of a multiplication CMS gadget [12]

In the following section some definitions, used to qualify the security of a circuit, will be given.

Definition 2.8 (d-Probing Security)**.** *A circuit **C** is **d-Probing Secure** iff every combination of q shares is independent of the corresponding sensitive variable.*

An additional level of security is given by the d-Non Interference:

Definition 2.9 (d-Non Interference (d-NI))**.** *A circuit **C** is **d-Non Interfering** iff, given i probes on internal wires and o probes on the outputs of **C**, an attacker can't obtain information on more than $i + o$ shares of any sensitive variable, where $i + o \leq d$.*

Verifying the security of an entire circuit is often very time consuming and complicated. Therefore smaller portions of the analyzed circuit are usually considered and combined later on. These subsections of circuits are going to be referred to as *gadgets*. When combining two t-NI secure gadgets, the outputs of the first become inputs for the second one, as can be seen in Figure 2.13. In this example an attacker might be able to bypass the protection of the two 2-NI secure gadgets, by placing only three probes (in positions a, b and c) instead of four. This is due to the fact, that the probe in position b can be used as output probe for gadget A, as well as input probe for gadget B. Therefore an even stronger security definition is needed in order to overcome this problem.

Definition 2.10 (d-Strong Non Interference)**.** *A circuit **C** is **d-Strong Non Interfering** iff, given i probes on internal wires and o probes on the outputs of **C**, an attacker can't obtain information on more than i shares of any sensitive variable, where $i \leq d$.*

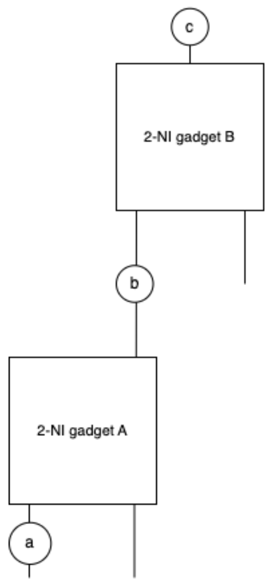


Figure 2.13: Combination of two NI gadgets

Robust probing security

All the definitions given in the previous Section consider ideal circuits in which no physical defaults are present. In order to model glitches, transitions and couplings (see Section 2.4.2) the following models for *extended probes* are introduced:

Definition 2.11 (Extended probes for combinatorial recombinations). *Given a n-input circuit* $\boldsymbol{C}$*, combinatorial recombinations (or glitches) can be modelled with n-extended probes, so that probing an output of* $\boldsymbol{C}$ *allows an attacker to also obtain the values of all its n inputs.*

Definition 2.12 (Extended probes for memory recombinations). *Given a memory cell* $\boldsymbol{m}$*, memory recombinations (or transitions) can be modelled with 2-extended probes, so that probing m allows an attacker to also obtain the value contained in* $\boldsymbol{m}$ *in the previous invocation.*

Definition 2.13 (Extended probes for routing recombinations). *Given a set of n adjacent wires* $\mathcal{W} = \{w_0, ..., w_{n-1}\}$*, routing recombinations (or couplings), can be modelled with d-extended probes, so that probing on one wire* w_i *allows an attacker to obtain the values on d adjacent wires.*

Considering the just defined extended probes, two stronger security definitions are given:

Definition 2.14 (d-robust Non Interference (d-robust NI)). *A circuit* $\boldsymbol{C}$ *is* ***d-robust Non Interfering*** *iff, given i extended probes on internal wires and o* *extended probes on the outputs of* $\boldsymbol{C}$*, an attacker can't obtain information on more than* $i + o$ *shares of any sensitive variable, where* $i + o \leq d$*.*

Definition 2.15 (d-robust Strong Non Interference (d-robust SNI))**.** *A circuit* $\boldsymbol{C}$ *is* ***d-robust Strong Non Interfering*** *iff, given* i *extended probes on internal wires and* o *extended probes on the outputs of* $\boldsymbol{C}$*, an attacker can't obtain information on more than* i *shares of any sensitive variable, where* $i \leq d$*.*

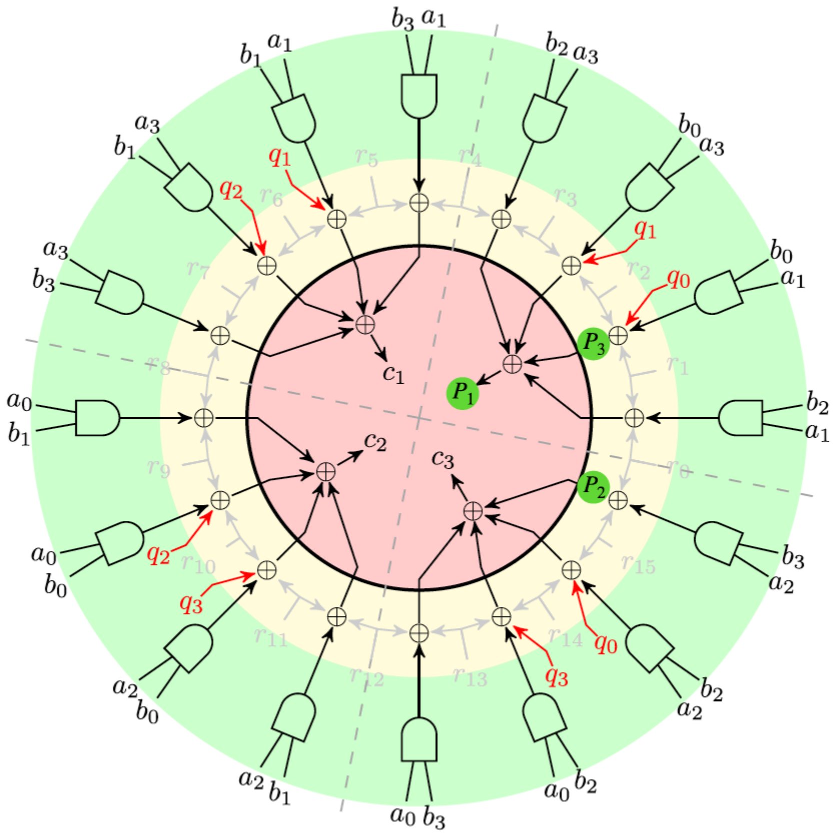


Figure 2.14: CMS-like 3-SNI but not robust 3-SNI gadget [28]

Figure 2.14 shows a 3-SNI CMS implementation with 4 shares.
Positioning three standard probes in the positions marked in green by P_1, P_2 and P_3 the information that can be obtained are $([a_1b_2 + r_0 + r_1] + [a_1b_0 + r_1 + r_2 + q_0] + [a_3b_0 + r_2 + r_3 + q_1] + [a_3b_2 + r_3 + r_4])$ from probe P_1, $(a_2b_3 + r_0 + r_15)$ from probe P_2 and $(a_1b_0 + r_1 + r_2 + q_0)$ from probe P_3. In this case all shares of the two sensitive variables a and b are protected by at least two random bits, and an attacker can't obtain information on more than 2 shares of any sensitive variable. This gadget is therefore 3-SNI.
On the other hand positioning three extended probes P_1, P_2 and P_3 in the same

positions, an attacker can get $(a_1b_2 + r_0 + r_1, a_1b_0 + r_1 + r_2 + q_0, a_3b_0 + r_2 + r_3 + q_1, a_3b_2 + r_3 + r_4)$ from the external probe P_1, (a_2b_3, r_0, r_15) from internal probe P_2 and (a_1b_0, r_1, r_2, q_0) from the internal probe P_3. By adding r_0, obtained from P_2, and r_1, obtained from P_3, to the information obtained from P_1, the attacker has access to three shares of b, using only 2 internal probes. Share b_2 is exposed by probe P_1, share b_3 by probe P_2 and b_1 is exposed by probe P_3. Therefore this gadget is not robust 3-SNI.

2.5 Verilog

In order to analyse physical implementations of algorithms a description of the resulting circuit is needed. For this purpose, in this thesis the hardware description language *Verilog* is used.
Verilog, released in 2015, is mainly used for the design and validation of digital circuits at a *register-transfer level* (RTL) of abstraction. At this level of abstraction, synchronized circuits are represented as signal flows between registers and the logical operations performed on those signals.

Chapter 3

Tools

3.1 Exploited tools

Several tools have been used for the different stages of the simulations made during this work. In the following chapter a brief introduction for every one of those tools is given.

3.1.1 OpenLane

OpenLane is a tool, which, given a description in Verilog of a circuit, returns the *GDS2* (or *GDSII*) file of that circuit [1]. A GDS2 file is a binary description of the layers of a circuit, which are used by the foundry for the production. This process is usually used to obtain the GDS2 description of *macros*, and is therefore called *macro hardening*. Macros are smaller portions of circuits, performing specific functionalities, which can be combined to produce the desired chip.

In order to generate this files, OpenLane performs six steps:

- *Synthesis:* During this step, OpenLane generates the *netlist* of the circuit, passed as input. The netlist of a given circuit contains a list of all its components and a description of how these components are interconnected. The components are taken from a library of standard cells contained in OpenLane. Synthesis starts from a logical description of the operations a circuit has to perform and returns the physical structure the circuit has to have, in order to perform them.
  The synthesis step is performed by an external tool called *yosys*. After the netlist has been generated, a tool called *OpenSTA* performs a static timing analysis on the obtained netlist.

- *Floorplanning:* This step is necessary to calculate the space needed to fit all components, found in the previous step, onto the chip.

- *Placement:* After having calculated the surface of the chip, the different components have to be placed in the right position. This is done during the placement stage, by performing first a *coarse placement*, and then a *fine placement* step. In the first step all components are placed roughly in the correct place. Only in the fine placement the final position is found. During this step OpenLane will also make sure that there is no overlapping between any two components.

- *Clock Tree Synthesis (CTS):* The clock is one of the most important signals in the circuit, since it is used by a large number of gates. During this step, OpenLane ensures that this signal arrives, with the needed strength, to each one of those gates.

- *Routing:* After the clock signal has been connected in a correct way, OpenLane continues by connecting the power supply and the ground to the corresponding pins of the components.

- *Signoff:* In this step, some final checks on the obtained results are performed, before sending out the obtained GDS2 file.

During the CTS and Routing step OpenLane modifies the initially generated netlist, according to the newly arisen needs. Therefore a *Logic Equivalence Check (LEC)* is performed after both steps, using yosys [15]. As the name suggests, the objective of this step is to check whether the new netlist is functionally equivalent to the originally synthesized netlist.
In order to generate the GDS2 file, OpenLane makes also use of a *Process Design Kit* (PDK). A PDK is a foundry specific set of files, which are used during the design of the circuit, containing for example the checks, performed during the signoff stage, or the cell library used by yosys during the synthesis step.

OpenLane for VoLPE

In order to use OpenLane for our tool, the following steps have to be executed (see Listing 3.1 for the commands that have to be executed for each step).

1. A folder, which is going to contain the source Verilog files of the circuits, that need to be synthesized, has to be created in the main folder of OpenLane. This is a optional step, which simplifies finding the needed files in later steps.

2. In the *designs* folder of OpenLane, a new folder for the current design has to be created. In order for OpenLane to find this folder, the source code and the main module of the design, they have to have same name. Given for example a source code file called *aes_sbox.v*, then the main module inside that code, as well as the design folder, have to be called *aes_sbox*.

3. Back in the OpenLane root folder the docker container has to be started.
4. The *flow.tcl* script, contained in the docker has to be executed, which inserts the design source code in the previously created design folder.
5. In addition to inserting the source code in the corresponding folder, the flow.tcl also creates a configuration file for the design. This file has to be modified, making sure that the clock name specified in the configuration file corresponds to the name, given to the clock in the main module of the design.
6. Using the same *flow.tcl* the circuit can be synthesized. A specific tag can be given to each synthesis run.
7. The result can be found in the

$$/OpenLane/designs/design/runs/tag/results/final/verilog/gl/ \tag{3.1}$$

   folder, and need to be copied into the *scr* folder of the VoLPE project.

Listing 3.1: Commands necessary for the usage of OpenLane

```
// Step 1
cd OpenLane
mkdir my_designs
cd my_designs
cp path/to/<design>.v

// Step 2
cd OpenLane/designs
mkdir design

// Step 3
make mount

// Step 4
flow.tcl -design <design> -src my_designs/<design>.v -
    init_design_config -add_to_designs

// Step 6
flow.tcl -design <design> -tag <tag>
```

3.1.2 Icarus Verilog

To analyse the behaviour of a circuit, given specific inputs, the circuit can be simulated. For this purpose, the *Icarus Verilog* tool can be used. Icarus compiles the Verilog source code, generating a *.vvp* file, which can then be executed to start

the simulation. In addition to the circuit that needs to be simulated, Icarus also needs a *testbench*, which is a file that has to be generated by the user. With this file the user enters the sequence of input values that Icarus has to simulate.
During the simulation a *.vcd* file is generated, which can then be used by a wave viewer, such as *GTKWave*, to display the value each wire of the circuit assumes during the simulation.

3.2 Tools for probing security analysis

3.2.1 IronMask

The manual verification of security properties has been shown to be very error-prone already on small gadgets. Automatic tools have therefore been introduced, to apply formal verification methods on these gadgets and to qualify their security.
IronMask is an automatic verification tool, which can check a circuit for all probing security properties discussed in Section 2.4.4 [3]. These checks are performed using an internal functions called *sets of input shares* (SIS) function. This function takes as inputs a high level description of the gadget and a set of probes placed on that gadget. Performing a number of operations on the probes, it is able to determine the input shares that are leaked.
Several optimizations are included in the tool, in order to make the verification faster.
As other currently available automatic verification tools, IronMask performs its computations on a high level description of the gadget. This high level description does not take the physical implementation of that gadget into consideration. The presence of glitches and other physical defaults in the gadget can therefore only be assumed. It is up to the user of the tool to decide, during the verification, if the placed probes should be considered extended or not (see Section 2.4.4 for the definition of extended probes).

Chapter 4

Analysis of circuit security

Once a secure cryptographic algorithms has been developed, it is implemented into a physical circuit, in order to carry out its specific tasks. As discussed in Section 2.4, this opens up the possibility for attackers, to exploit Side Channel Attacks, to break that algorithm and find the secret key handled by it. It is in the best interest of who produces such circuits to know if their product is secure, before sending it into production.
For this reason, the probing models discussed in Section 2.4.4 where introduced. However these models don't take the actual structure of the circuit into consideration.

The goal of this work, is to generate a tool, which can analyse the weakness of a circuit towards SCA, as realistically as possible. The developed tool, called VoLPE (Verification of Leakages Propagation Escalation), does that, by simulating a synthesized circuit, obtained from OpenLane, and returning the existing correlation value, between the inputs and a model, describing the power consumption of the circuit.
The code for VoLPE can be found at the following link:
`https://github.com/LorenzoGiacobbe/CodiceTesiMagistrale`.

4.1 Structure of the developed tool

A general structure of the tool can be observed in Figure 4.1. After having generated a high level description of the tool with Verilog, the user has to obtain the synthesized version of the gadget using OpenLane. The Verilog file containing the synthesized description of the gadget is then passed as input to VoLPE, together with a configuration file, also created by the user. The *sim.sh* script inside VoLPE performs the simulations of the gadget, returning the generated simulation logs to the main module. Using these logs VoLPE calculates the required results, which are then saved in an Excel file.

In the following Section the main components of the tool are analysed and described more in detail. We first start by describing the meaning of the mandatory fields contained in the configuration file. Proceeding then with an explanation of how the gadgets are simulated and which intermediate results are generated. Finishing this section with a description of the models used for the calculation of the final results.

From now on, with the term *circuit*, the synthesised version obtained after the execution of OpenLane is intended.

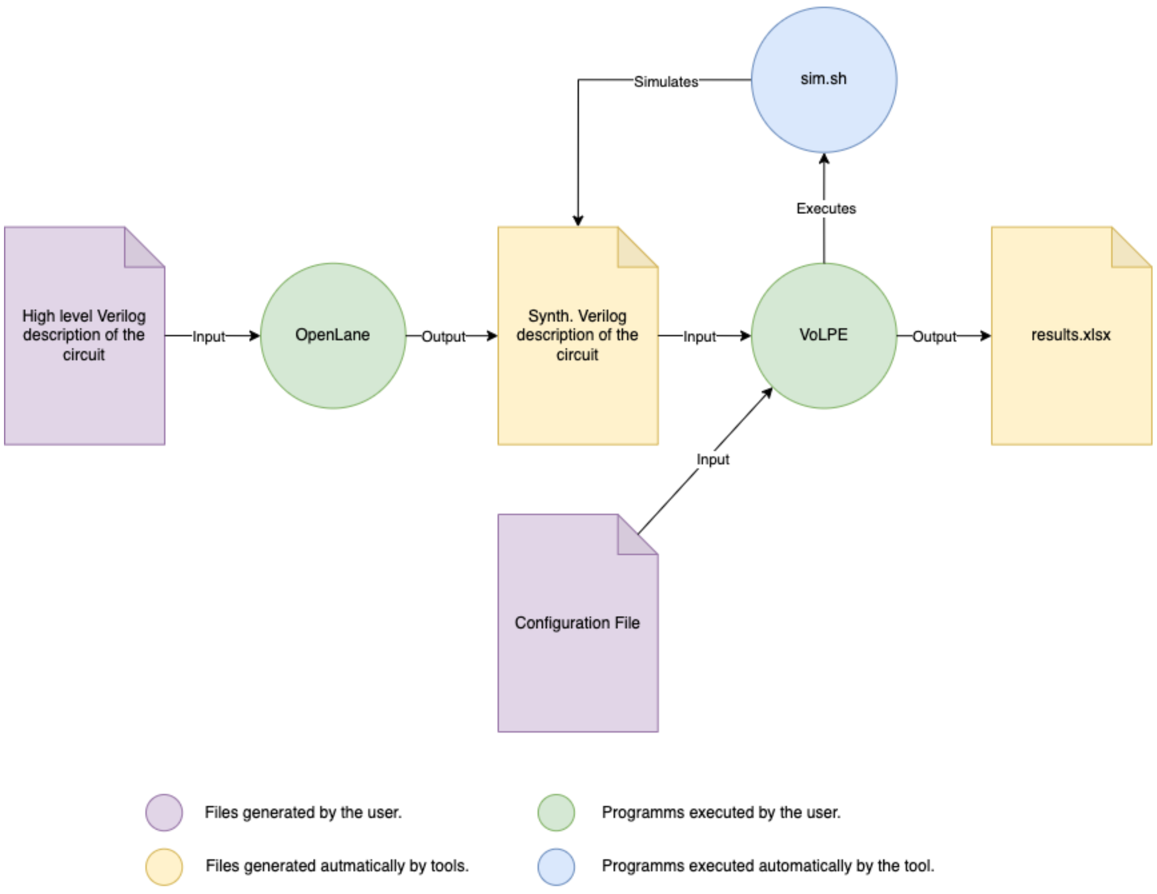

Simulates
sim.sh
Executes
High level Verilog description of the circuit
Input
OpenLane
Output
Synth. Verilog description of the circuit
Input
VoLPE
Output
results.xlsx
Input
Configuration File
Files generated by the user.
Programms executed by the user.
Files generated autmatically by tools.
Programms executed automatically by the tool.

Figure 4.1: Structure of the developed tool.

4.1.1 Configuration files

Before starting the tool, some basic information about the circuit that has to be simulated have to be specified. This is done through a configuration file, where the following fields have to be filled:

- *full:* As will be explained with more detail in the next Section, it is possible to simulate the analyzed circuit exhaustively or only partially. By initializing this field with the value y, the exhaustive simulation is executed. For a partial simulation, the value n has to be given to this field.
- *sim:* With this field the user can specify how many simulations the tool will perform. This number can be calculated as follows: $sim = 2^{2*x}$, where x has to be equal to in_size (see below for a description of this value), if the previous field was set to y. Whereas, if *full* was set to n, then x can be any whole number between 0 and in_size. The number of simulations that have to be performed needs to be specified even when *full* is set to y.
- *clk:* With this field the presence or absence of a clock in the simulated circuit can be indicated. The value y has to be assigned if a clock is needed, and the value n otherwise.
- *period:* Through this field the user can set the period of the clock that will be used during the simulations. In our tool the clock period defines also, after how much time the inputs are changed during the simulation. This value must be large enough, for the circuit to finish the calculations with the previous input, before starting with the new input of the following cycle, to prevent undefined behaviour. If for example the circuit needs $5ns$ to propagate the input signal, then the value of *clk* should be strictly greater than that value, in order to allow the new state to settle, before the next input is entered. In order to calculate maximal propagation time, the slowest path inside the circuit, has to be considered. To define the slowest path, for every possible path the input can cross in the circuit, the delays of all gates, on that path, have to be summed. The path with the greatest total delay, is the slowest one.
  Even if the analysed circuit has no clock, this field needs to be defined, and will represents the time the tool waits to change the inputs of the circuit.
- *cycles:* The *cycles* field indicates, how many clock cycles the circuit needs, to perform its calculations. This value has to be equal to the number of register levels in the circuit, plus one.
- *in_size/out_size/rand_size:* These fields indicate the number of input, output and random bits of the circuit.
- *in_name/out_name:* Here the name of the variables indicating the input and output bits of the circuit are specified.
- *input delayed_input_name:* With these fields, it is possible to indicate an arrival delay for each input. An absent delay has to be indicated with the value 0.

- *gate delayed_gate_name:* As for the previous field, this one has to be used to define the delays, introduced by each kind of gate in the circuit.

In Listing 4.1 a sample configuration file can be seen.

Listing 4.1: Sample configuration file

```
sim: 576
full: n
clk: y
period: 5
cycles: 2
in_size: 6
in_name: in
rand_size: 2
out_size: 4
out_name: out

input a0     0.02
...

gate XNOR_DELAY   0.10
gate NAND_DELAY   0.20
gate XOR_DELAY    0.50
gate AND_DELAY    0.30
gate OR_DELAY     0.40
```

4.1.2 Simulation

After having discussed the steps that need to be performed before starting the tool, the actual simulation of the circuit can be analyzed.

Definition 4.1 (State)**.** *Given a circuit C, its state is given by the values assumed by its output bits.*

Definition 4.2 (Simulation)**.** *Let C be a circuit with an initial state S_i. During a simulation, a new input i is entered and a change in the value of the output can be detected. The simulation ends when the new state reaches a stable final value S_f.*

The first step performed by the tool is the generation of the inputs and initial states for each simulation, which are then saved on a dedicated file.
As mentioned earlier, the circuit can either be simulated exhaustively or partially.

Exhaustive simulation

The first aim is to simulate the circuit with all the possible input combinations.

Example 4.1. *Given a circuit, performing the AND operation of two inputs, the four possible initial states are:*

$$\mathbf{S} = \{(0,0), (0,1), (1,0), (1,1)\} \tag{4.1}$$

All the possible two bit combinations of its inputs are:

$$\mathbf{I} = \{(0,0), (0,1), (1,0), (1,1)\} \tag{4.2}$$

In order to perform an exhaustive simulation, starting from each initial state of **S**, *every input from* **I** *has to be entered (see Table 4.1).*

$State_i$	$Input_i$
(0,0)	(0,0)
(0,0)	(0,1)
(0,0)	(1,0)
(0,0)	(1,1)
(0,1)	(0,0)
(0,1)	(0,1)
(0,1)	(1,0)
(0,1)	(1,1)
...	...
(1,1)	(0,0)
(1,1)	(0,1)
(1,1)	(1,0)
(1,1)	(1,1)

Table 4.1: All possible simulations for Example 4.1

Partial simulation

The number of simulations, however, grows exponentially with each additional input bit added to the circuit. This leads to a considerable increment in computation time for the tool. In order to allow faster executions, the possibility to perform only a subset of all possible simulations has been implemented. In this mode, the initial states and the inputs that are going to be simulated are generated randomly. Python's *os.urandom* function is used for the random number generation, which is suitable for cryptographic applications. Since not all possible initial states and inputs are tested, the results here obtained are less realistic.

Runs of simulation

Once the initial states and inputs have been generated, the actual simulation of the circuit, is performed. For every circuit it is possible to perform three different runs of simulations:

- *Circuit with no delays:* During this run, the circuit is considered to be "ideal", and therefore all delays are considered to be zero. Since no delays are present in this circuit, no glitches can occur during these simulations. Therefore, all found correlations are caused by the algorithm itself and not by the structure of the circuit.
- *Circuit with gate delays:* During this run, the simulation takes gate delays into consideration. With the addition of delays some glitches might appear. These runs simulate a circuit which receives synchronized inputs, which arrive with no delays.
- *Circuit with gate and input delays:* During this run, some delays to the inputs are added. These runs simulate the case, in which the inputs to the circuit are not generated synchronously and can therefore arrive with non-zero delays.

All the steps discussed in the next sections are executed in the same way for all three runs.

Accepted Circuits

In order to perform the simulations, Icarus executes a *testbench* file. The testbench internally creates an instance of the circuit, it has to simulate, retrieves the initial states and inputs and starts the simulations of the instantiated circuit. In the current version of the tool the testbench accepts only circuits, which have the following ports:

- Input array of arbitrary length
- Output array of arbitrary length
- Clock (optional)

The structure of a circuit that can be simulated can be seen in Listing 4.2.

Listing 4.2: Accepted circuit structure

```
module circuit(in, out, clk);
    input [in_size:0] in;
    input clk; //optional
    output [out_size:0] out;
    ...
endmodule
```

Testbench

The testbench performs the following steps:

1. Before starting each simulation, the testbench reads the inputs needed to set the circuit into the correct initial state. These inputs are read from the previously generated input file.
2. It then passes the read inputs, to the circuit (see line 4 in Listing 4.3) and waits for the state to settle. The waited time is given by the clock period, multiplied by the number of cycles that the circuit needs, to finish its calculations.
3. When the initial state is set, it starts the simulation. It does so by reading the new input value from the input file and passing it to the circuit (see Line 6 in Listing 4.3).
4. At this point the state is monitored and each of its changes, is saved in a specific log.
5. The state's changes saved in this log will later be used to generate a power consumption model for this simulation.

Listing 4.3: Testbench

```
1 inputs = get_inputs("inputs.dat")
2 states = get_states("inputs.dat")
3 for i range(number_of_simulations):
4     in = state[i]
5     wait(clk_period * cycles)
6     in = input[i]
7     monitor(out, log)
```

4.1.3 Power consumption model

Once all simulations have been run, the actual power consumption has to be extracted from the previously generated logs. For each simulation, the power consumption is modelled through the number of toggles performed by the output, before settling to the new state. This step generates a list (tl), containing the number of toggles the state performs during each simulation.

Definition 4.3 (Toggle)**.** *A* ***toggle*** *identifies the change of a bit, from the value* 0 *to the value* 1 *or vice-versa.*

Example 4.2. *A change of an array of bits* $\mathbf{S} = \{b_0, b_1, b_2, b_3\}$ *from the values* $\mathbf{S} = \{0,1,1,1\}$ *to the values* $\mathbf{S} = \{1,1,0,1\}$*, implies that two toggles have occurred. The first being* b_0 *switching from* 0 *to* 1 *and the second one being* b_2 *switching from* 1 *to* 0*.*

This model was deemed appropriate, for this work, since the toggles of the state bits represent a power consumption model close to reality.

4.1.4 Consume model

After having generated a power consumption model for each simulation, a model describing the operation, which caused that power consumption has to be found. We decided to generate two different values using two different functions:

1. *Input consume model (I-CS):* This model is used to find the correlation between the power consumption and the inputs that generated it.

2. *Input-state consume model(IS-CS):* This model is used to find the correlation between the power consumption and the combination of initial state and input that generated it.

Since the user may consider only a part or a certain combination of the input and initial state bits to be of interest for the models, we implemented a selection function for each of the two models. This function allows the user to select the relevant bits for the creation of each consume model. Once the relevant bits have been chosen, a second function creates the corresponding models.

Although these functions can be redefined by the user, in the following sections the implementations proposed by us are discussed with more detail.

Selection function

For both consume models we implemented a selection function, which groups the input and initial state bits, into groups of n adjacent bits:

- *Input selection function:* This function, whose pseudocode is shown in Listing 4.4, accepts two arguments: a list il, containing the input bits $i_x = \{i_0, ..., i_{m-1}\}$ of each simulation, and an integer n, which represents the dimension of the desired groups.
  The input selection function returns m/n lists. For every element i_x of il, the corresponding element $i_{y,x}$ of each newly created list il_y consists of the y-th group of n bits of i_x: $i_{y,x} = \{i_{(y-1)*n}, ..., i_{(y*n)-1}\}$.
  Considering for example a circuit with $m = 4$ inputs and a desired size for the groups $n = 2$, the selection function creates $4/2 = 2$ lists, il_1 and il_2. For every element $i_x = \{i_0, ..., i_3\}$ of il, the corresponding values of il_1 and il_2 are $i_{1,x} = \{i_0, i_1\}$ and $i_{2,x} = \{i_2, i_3\}$, respectively.

- *Input-state selection function:* This function, whose pseudocode is shown in Listing 4.5 accepts, in addition to the arguments of the input selection function, a third argument: a list sl, containing the initial state bits $s_x =$

$\{s_0, ..., s_{m-1}\}$ of each simulation. The input-state selection function performs the same steps as the previous selection function, not only on il but also on sl, and returns $2 * (m/n)$ lists.

To guarantee the correct functioning of the selection functions m must be a divisor of n.

Listing 4.4: Input selection function

```
input_selection_function(il, n):
    size = m / n

    for i in range(len(il)):
        for input in range(size):
            for c in range(n):
                r += il[i][input*n+c]
            result_il[input].append(r)

    return result_il
```

Listing 4.5: Input and state selection function

```
input_state_selection_function(il, sl, n):
    size = int(m / n)

    for i in range(len(il and sl)):
        for input in range(size):
            for c in range(n):
                r_il += il[i][input*n+c]
                r_sl += sl[i][input*n+c]
            result_il[input].append(il)
            result_sl[input].append(sl)

    return result_il, result_sl
```

Consume model

After having selected the desired bits with the selection functions the two consume models are generated:

- *Input consume model:* This function calculates the hamming weight for each element of the lists returned by the input selection function.
  For example, given a circuit with 4 inputs and $n = 2$, the input consume model receives two lists from the input selection function, whose i-th elements contain $\{i_{i,0}, i_{i,1}\}$ and $\{i_{i,2}, i_{i,3}\}$ respectively. For each one of those lists a new list is created by the input consume model, whose i-th elements are $m\_hw_{1,i} = hw(\{i_{i,0}, i_{i,1}\})$ and $m\_hw_{2,i} = hw(\{i_{i,2}, i_{i,3}\})$ respectively.

- *Input-state consume model:* This function calculates, for each pair of sl_x and il_x, the hamming distance between each of their corresponding elements.
  For example, given a circuit with 4 inputs and $n = 2$, the input-state consume model receives four lists from the input-state selection function, whose i-th elements contain $il_1 = \{i_{i,0}, i_{i,1}\}$, $il_2 = \{i_{i,2}, i_{i,3}\}$, $sl_1 = \{s_{i,0}, s_{i,1}\}$ and $sl_2\{s_{i,2}, s_{i,3}\}$. The two lists created by the input-state consume model are m_hd_1 and m_hd_2, whose i-th elements are $m\_hd_{1,i} = hd(\{i_{i,0}, i_{i,1}\}, \{s_{i,0}, s_{i,1}\})$ and $m\_hd_{2,i} = hd(\{i_{i,2}, i_{i,3}\}, \{s_{i,2}, s_{i,3}\})$.

4.1.5 Correlation matrix

During this last step the Pearson's correlation is calculated, between the power consumption model (tl) and each consume model (m_hw_i) and (m_hd_i):

- $pearsonCorrelation(tl, m\_hw_i)$, $\forall i \in [0, m/n]$
- $pearsonCorrelation(tl, m\_hd_i)$, $\forall i \in [0, m/n]$

An example correlation matrix can be seen in Table 4.2.

	m_hw_1	m_hd_2	m_hd_1	m_hd_2
no delays	0	0	0.1252	0.1252
gate delays	0,0008	0,0796	0,0195	0,1263
input and gate delays	0,0004	0,0456	0,5853	0,6611

Table 4.2: Sample correlation matrix for a circuit with 8 inputs and $n = 4$

Chapter 5

Testing and results

After having given a description of the developed tool in Section 4.1, in this Section we summarize the results obtained using the tool.
We decided to start by testing our tool on some unprotected implementations of the cryptographic algorithms AES and Xoodoo, in order to verify the intrinsic security of these algorithms against SCAs. We focused in particular on the non-linear parts of both algorithms, namely the `SubBytes` function of AES and the χ function of Xoodoo.
Finally we tested the vulnerability of the two masking schemes CMS and DOM, described in Section 2.4.3 and Section 2.4.3, against SCAs.
Several implementations have been tested for each gadget, which are described more in detail in the following sections.

5.1 Delay selection

For all gadgets the first step was to find the worst possible combination of delays, which is the combination that maximizes the number of glitches. To do this we analyzed the synthesized circuits and fixed the delays based on their structure. We did this by choosing the delays in such a way, that as many sequences of gates as possible could generate glitches. Given for example a circuit section as in Figure 5.1, by choosing the delays in such a way that $delay\ A > delay\ C$ and $|delay\ B - delay\ C| > delay\ D$, both gates C and D will generate glitches. Whereas choosing a combination of delays such that $delay\ A < delay\ C$ and $|delay\ B - delay\ C| > delay\ D$, only gate D will generate glitches.

While this approach allowed to find significant delay values for CMS and DOM, it didn't turn out to be reliable for the more complex gadgets. Given the size of the code it was unfeasible to reconstruct in a correct way the structure of the gadget. We therefore performed the following steps in order to obtain the desired delays:

1. We executed 50 simulations for every gadget, using different input groups and

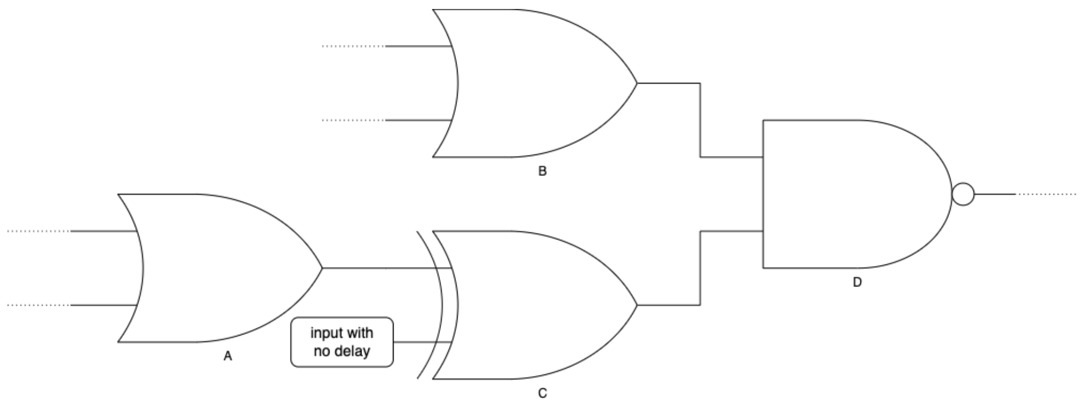


Figure 5.1: Sample circuit section.

random delay values.

2. With the data from the obtained correlation matrices we generated two summarizing matrices:
    - A matrix containing the maximal correlation value for all the simulations.
    - A matrix containing the average of the correlation values for all the simulations.
3. Once we found the simulation run with the highest correlation, we extracted the used delays.

5.2 AES S-Box

For the S-Box of AES we decided to perform our tests on two different implementations. The first implementation uses a lookup table to do the byte substitution, following the process described in Section 2.2.1. The second implementation uses a sequence of MUX gates, to perform the same operation. The second implementation is able to perform not only the encryption but also the decryption operation. The operation it performs is decided by setting a configuration bit to 0 for encryption or 1 for decryption. By passing it to OpenLane once with this bit set internally to 0 and once without setting it, we obtained two different synthesized circuits. By fixing the value of that bit to 0, OpenLane understands that the decryption part is never accessed and removes it from the synthesized circuit. We therefore decided to test our tool on both versions of this implementation, resulting in three different simulated circuits. For every circuit we calculated the correlation first for every input bit taken singularly and then for all input bits grouped together (see Section 4.1.4).

5.2.1 AES S-Box lookup table

AES-Sbox Version 1 Max. correlation values				
	1 bit I-CS	**8 bit I-CS**	**1 bit IS-CS**	**8 bit IS-CS**
no delays	0,0000	0,0000	0,0211	0,0345
gate delays	0,0811	0,0974	0,1334	0,1692
gate+inputs delay	0,1128	0,0485	0,3836	0,8504

(a) Maximal correlation values for the AES S-Box lookup table implementation

AES-Sbox Version 1 Avg. correlation values				
	1 bit I-CS	**8 bit I-CS**	**1 bit IS-CS**	**8 bit IS-CS**
no delays	0,0000	0,0000	0,0211	0,0345
gate delays	0,0685	0,0722	0,1257	0,1614
gate+inputs delay	0,0470	0,0189	0,3236	0,7792

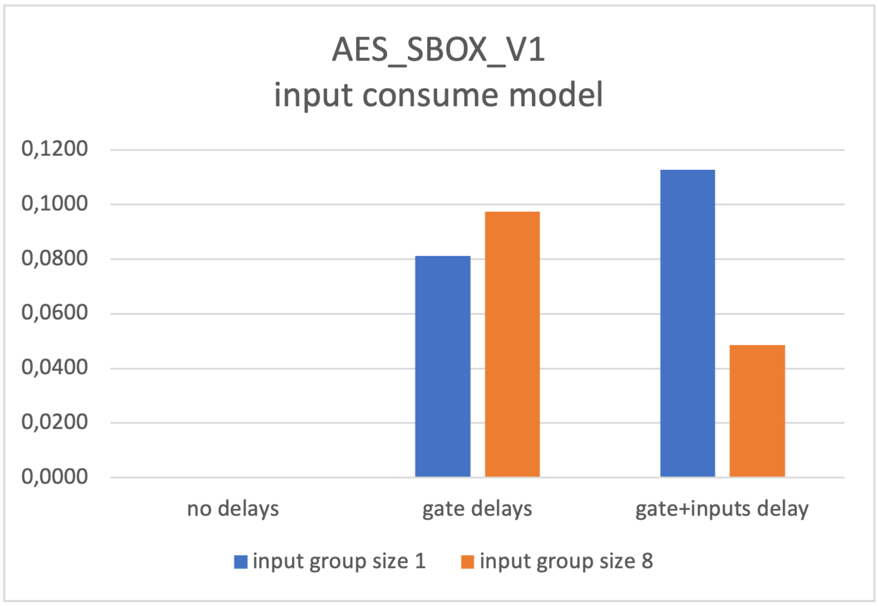
(b) Average correlation values for the AES S-Box lookup table implementation

Table 5.1: Average and maximal correlation values for the AES S-Box lookup table implementation

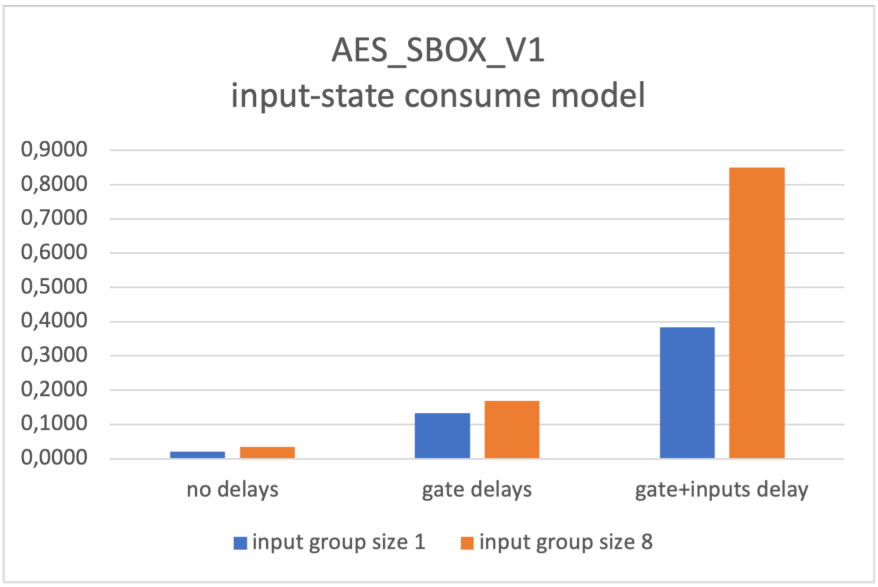
The first thing that can be noticed from Figure 5.1a is that the correlation of this circuit, when the input consume model is used and no delays are introduced, is zero. Showing how no additional information are leaked by this circuit, when considered "ideal" and when this consume model is used. A circuit is considered ideal when no delays are introduced during its simulation.

Comparing Table 5.2a, which shows the correlation for the various input groups, when using the input consume model, and Table 5.2b, which shows the same data for the input-state consume model, one can easily notice how the latter generates significantly higher correlation values, than the first one. While the highest correlation for the input consume model is 0.09, the correlation values for the input-state consume model reach up to 0.85.

The correlation values of the input-state consume model have a more than linear increase, when gate delays are introduced in the circuit. These values rise from 0.02 to 0.13, for a group of 1 bit, and from 0.03 to 0.17, when all bits are grouped together. The same is true when also input delays are added, making the correlation values rise from 0.13 to 0.38 and from 0.17 to 0.85, respectively for the two group sizes. These values also show how the correlation values visibly increase, when more bits are grouped together. When the 8 input bits are grouped together the correlation reaches a value, that is more than double the correlation reached when the input bits are considered singularly.



(a) Input consume model chart for the AES S-Box lookup table implementation



(b) Input-state consume model chart for the AES S-Box lookup table implementation

Figure 5.2: Input and input-state consume model chart for the AES S-Box lookup table implementation

When looking at the results of the input consume model, a less steep increase in correlation values can be noticed, when gate and input delays are added. By grouping all the input bits together the correlation value even decreases from 0.09 to 0.05, when input delays are added.

As can be seen from Table 5.1b the average correlation values are only slightly lower than the maximal correlation values, indicating how the used delays have a slim effect on the correlation.

The simulation run with the highest correlation values used the delays that can be seen in Tables A.3a and A.3b.

5.2.2 AES S-Box MUX only encryption

AES-Sbox Version 2 Max. correlation values				
	1 bit I-CS	**8 bit I-CS**	**1 bit IS-CS**	**8 bit IS-CS**
no delays	0,0000	0,0000	0,0211	0,0345
gate delays	0,1427	0,0512	0,2711	0,1266
gate+inputs delay	0,1170	0,0398	0,2977	0,3181

(a) Maximal correlation values for the AES S-Box MUX implementation performing only encryption

AES-Sbox Version 2 Avg. correlation values				
	1 bit I-CS	**8 bit I-CS**	**1 bit IS-CS**	**8 bit IS-CS**
no delays	0,0000	0,0000	0,0211	0,0345
gate delays	0,0370	0,0210	0,1262	0,0867
gate+inputs delay	0,0356	0,0197	0,1555	0,1645

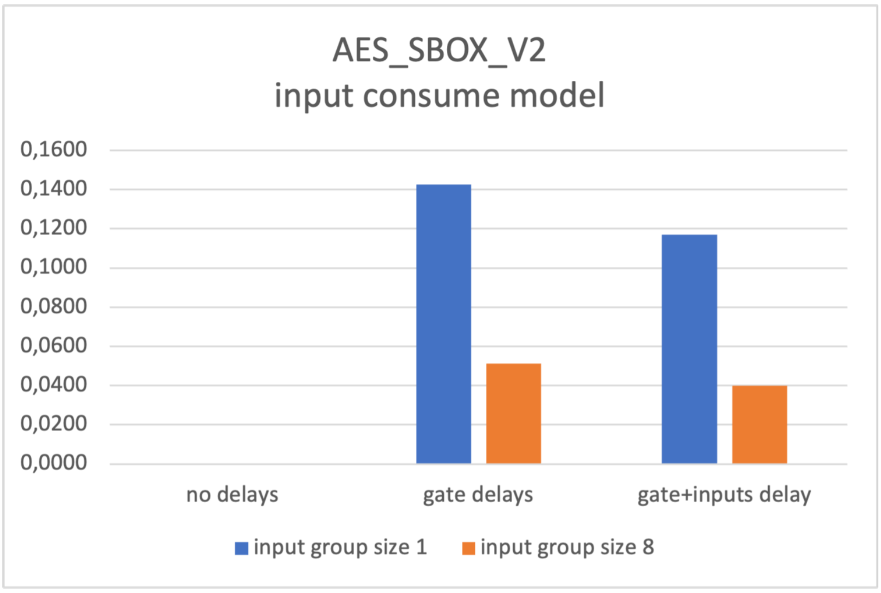
(b) Average correlation values for the AES S-Box MUX implementation performing only encryption

Table 5.2: Average and maximal correlation values for the AES S-Box MUX implementation performing only encryption

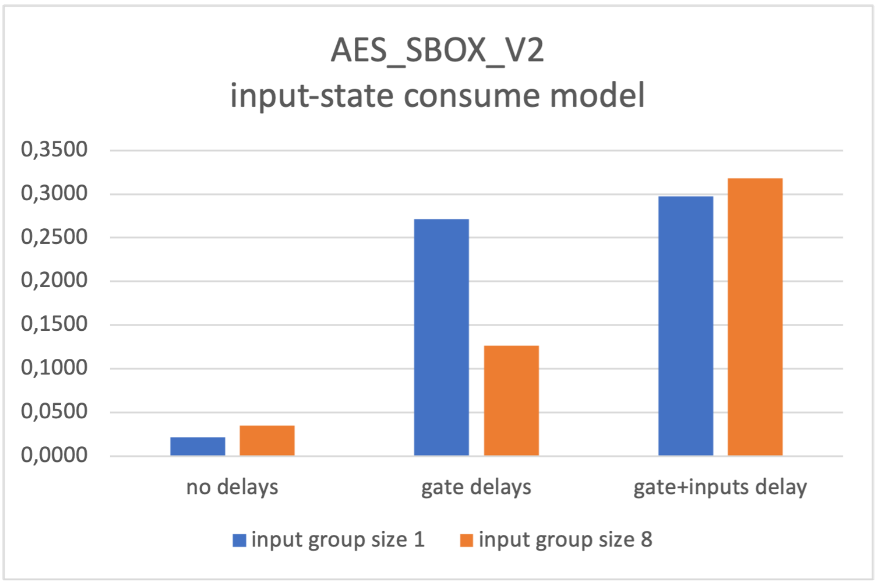
As for the previous implementation of the AES-Sbox the correlation values for the circuit without delays, when using the input consume model are zero. As for the previous gadget no additional information are leaked with this configuration.

Also for this implementation the use of the input-state consume model results in higher correlations, with respect to the input consume model.

In this implementation of the AES-Sbox the correlation values, when all the input bits are considered separately, show a big increase when gate delays are added, raising from 0.02 to 0.27. The introduction of input delays increases this



(a) Input consume model chart for the AES S-Box MUX implementation performing only encryption



(b) Input-state consume model chart for the AES S-Box MUX implementation performing only encryption

Figure 5.3: Input and input-state consume model chart for the AES S-Box MUX implementation performing only encryption

value only slightly from 0.27 to 0.30. On the other hand, when all input bits are grouped together the correlation values assume a similar trend to the previous case, starting at 0.03, raising to 0.13, when gate delays are introduced, and reaching their maximal value of 0.32, when also input delays are considered. While the correlation values show a similar trend as the implementation discussed in Section 5.2.1, the maximal correlation values are significantly lower. While the highest correlation for the first implementation was 0.85, this implementation only reaches a maximal correlation value of 0.32.

Differently from the first implementation, the average correlation values are only half of the maximal values, as can be seen from Table 5.2b. This indicates that the correlation values of this gadget can vary significantly depending on the chosen delay.

The delays resulting in the highest correlation can be seen in Tables A.5a and A.5b.

5.2.3 AES S-Box MUX encryption and decryption

AES-Sbox Version 3 Max. correlation values				
	1 bit I-CS	**8 bit I-CS**	**1 bit IS-CS**	**8 bit IS-CS**
no delays	0,0000	0,0000	0,0361	0,0470
gate delays	0,1221	0,1162	0,1532	0,1521
gate+inputs delay	0,1536	0,0511	0,3298	0,4529

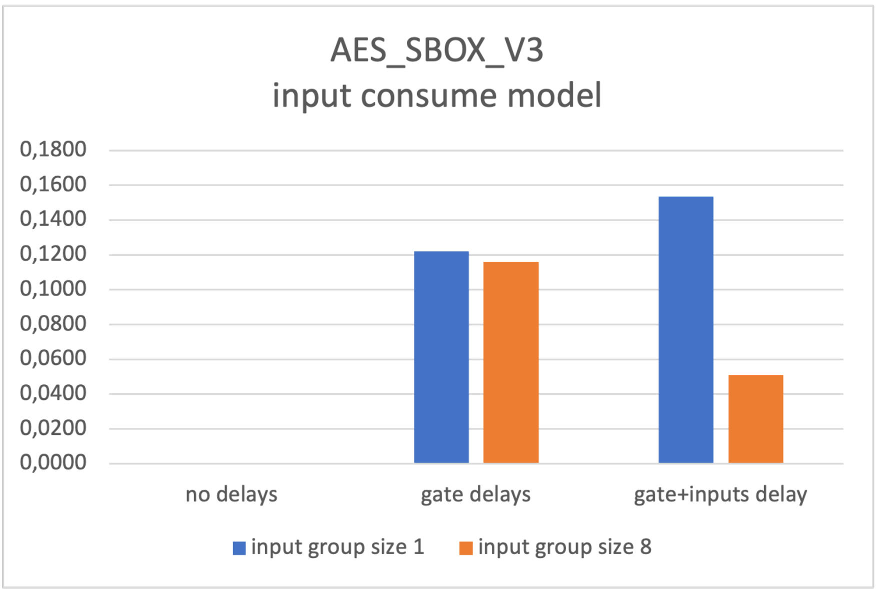
(a) Maximal correlation values for the AES S-Box MUX implementation performing encryption and decryption

AES-Sbox Version 3 Avg. correlation values				
	1 bit I-CS	**8 bit I-CS**	**1 bit IS-CS**	**8 bit IS-CS**
no delays	0,0000	0,0000	0,0228	0,0349
gate delays	0,0377	0,0187	0,0853	0,0950
gate+inputs delay	0,0339	0,0148	0,1931	0,1993

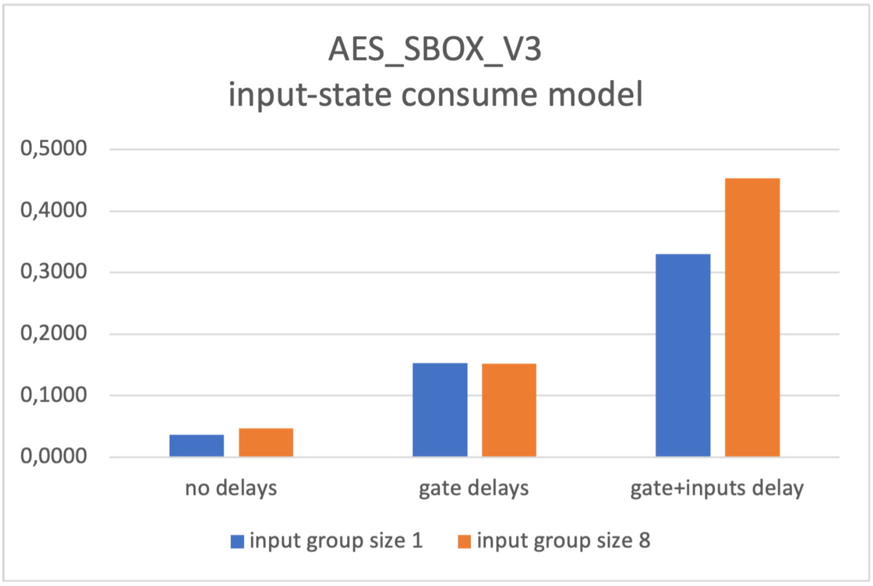
(b) Average correlation values for the AES S-Box MUX implementation performing encryption and decryption

Table 5.3: Average and maximal correlation values for the AES S-Box MUX implementation performing encryption and decryption

Also for this implementation the correlation values of the input consume model, for the circuit without delays, are zero.



(a) Input consume model chart for the AES S-Box MUX implementation performing encryption and decryption



(b) Input-state consume model chart for the AES S-Box MUX implementation performing encryption and decryption

Figure 5.4: Input and input-state consume model chart for the AES S-Box MUX implementation performing encryption and decryption

The computed correlations show high similarities with the first discussed implementation. The only differences being the lower maximal correlation value this implementation can reach and the higher difference between average and maximal correlation values. While the first version reaches correlation values of 0.85, the maximal correlation value of this implementation is 0.45. As for the second implementation of the AES-Sbox, the maximal correlation values are bigger than the average correlation values by a factor of more than 2. With respect to the second implementation the correlation values of this version are slightly higher: 0.32 for the second implementation and 0.45 for the third one. This is probably due to the fact that third circuit contains more gates, resulting in more opportunities for glitches to be caused.

The delays resulting in the highest correlation values can be seen in Tables A.5a and A.5b.

5.3 Xoodoo

This gadget was tested in two different versions. We tested first a basic version of the gadget, which does not contain a final level of registers. The second version was obtained by adding a layer of registers, through which the results are passed before being connected to the outputs. The considered implementations of Xoodoo take two values, both divided into 3 shares, resulting in six input bits. Therefore we decided to analyze the correlation of this circuit first by taking each bit individually, then by considering the input three bits at a time and finally by grouping all 6 bits together.

5.3.1 Xoodoo without final register layer

The first thing that can be seen from Table 5.4a is that the correlations, when only gate delays, or when gate and input delays are present, have the same values. This is due to the fact that this circuit contains an intermediate register level. As explained in Section 2.4.2 registers have the effect of cancelling out all the previously introduced delays; in this case the input delays. Therefore the introduction of input delays does not add any additional glitches, that can be seen at the output of the circuit. Although correlation values when using the input consume model are low, Figure 5.5a shows how the values decrease even more when grouping more bits together.

On the other hand when using the input-state consume model the correlation values increase, when the size of the input groups increases. When using this consume model, the correlations reach considerably higher values, reaching 0.86. For each group size a big difference can be noticed, between the correlation values of this circuit with no delays and the same circuit when delays are introduced.

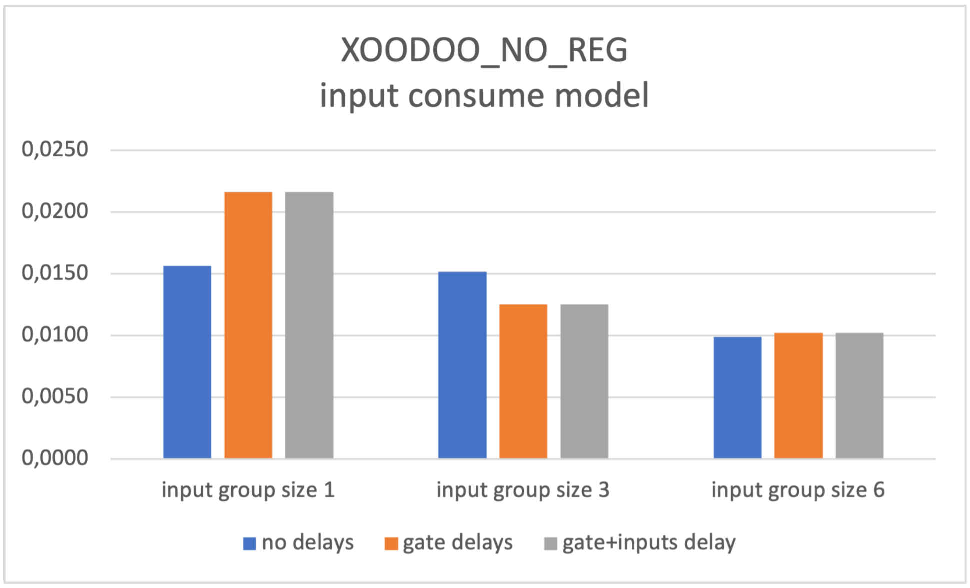
Xoodoo without registers Max. correlation values						
	1 bit I-CS	**3 bit I-CS**	**6 bit I-CS**	**1 bit IS-CS**	**3 bit IS-CS**	**6 bit IS-CS**
no delays	0,0157	0,0152	0,0099	0,1380	0,1973	0,2781
gate delays	0,0216	0,0125	0,0102	0,4373	0,6387	0,8564
gate+inputs delay	0,0216	0,0125	0,0102	0,4373	0,6387	0,8564

(a) Maximal correlation values for the Xoodoo implementation without final registers

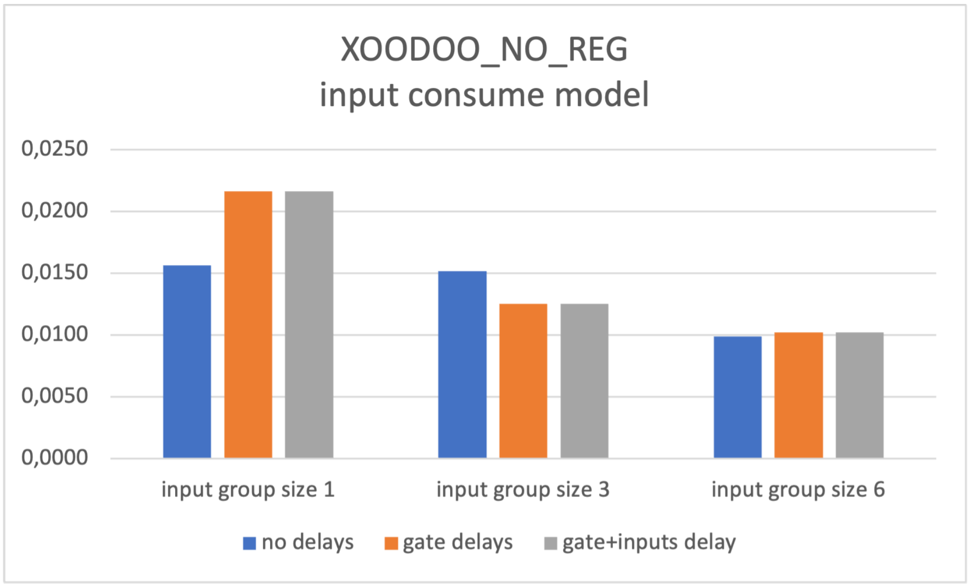
Xoodoo without registers Avg. correlation values						
	1 bit I-CS	**3 bit I-CS**	**6 bit I-CS**	**1 bit IS-CS**	**3 bit IS-CS**	**6 bit IS-CS**
no delays	0,0157	0,0152	0,0099	0,1380	0,1973	0,2781
gate delays	0,0148	0,0077	0,0053	0,3780	0,5446	0,7120
gate+inputs delay	0,0148	0,0077	0,0053	0,3780	0,5446	0,7120

(b) Average correlation values for the Xoodoo implementation without registers

Table 5.4: Average and maximal correlation values for the Xoodoo implementation without registers



(a) Input consume model chart for the Xoodoo implementation without registers



(b) Input-state consume model chart for the Xoodoo implementation without registers

Figure 5.5: Input and input-state consume model chart for the Xoodoo implementation without registers

As can be seen from Table 5.5b, the correlation values of the "ideal" circuit vary from 0.14 to 0.28. As soon as delays are introduced these values increase, ranging from 0.44, when each bit is considered singularly, to 0.86, when all inputs bits are considered as one group.

Comparing Tables 5.4a and 5.4b shows the small difference, between the maximal and average correlations, indicating that the correlation varies only slightly when different delays are considered.

The input and gate delays causing the maximal correlation value can be seen in tables A.7a and A.7b.

5.3.2 Xoodoo with final register layer

Xoodoo with registers Max. correlation values						
	1 bit I-CS	**3 bit I-CS**	**6 bit I-CS**	**1 bit IS-CS**	**3 bit IS-CS**	**6 bit IS-CS**
no delays	0,0040	0,0070	0,0099	0,1135	0,1967	0,2781
gate delays	0,0040	0,0070	0,0099	0,1135	0,1967	0,2781
gate+inputs delay	0,0040	0,0070	0,0099	0,1135	0,1967	0,2781

(a) Maximal correlation values for the Xoodoo implementation with registers

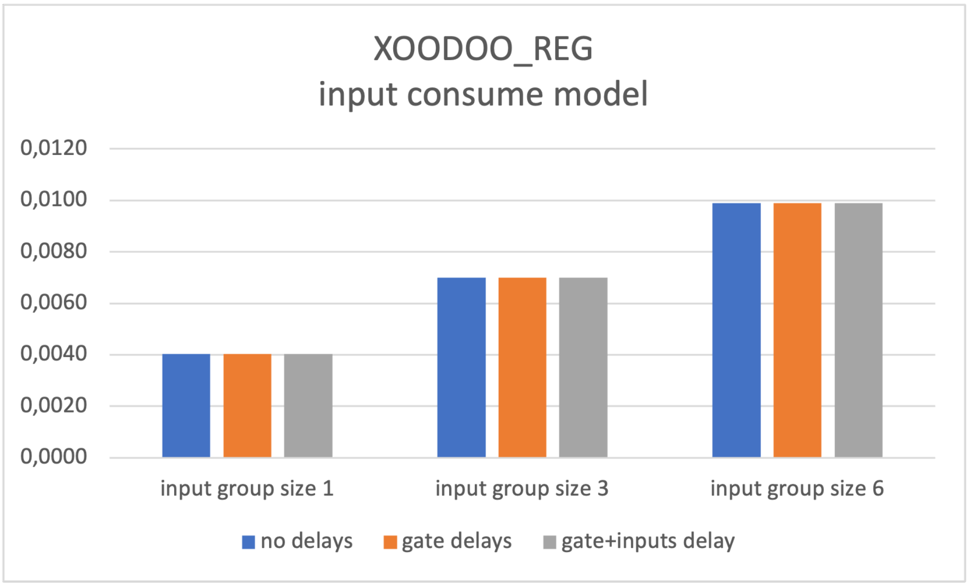
Xoodoo with registers Avg. correlation values						
	1 bit I-CS	**3 bit I-CS**	**6 bit I-CS**	**1 bit IS-CS**	**3 bit IS-CS**	**6 bit IS-CS**
no delays	0,0040	0,0070	0,0099	0,1135	0,1967	0,2781
gate delays	0,0040	0,0070	0,0099	0,1135	0,1967	0,2781
gate+inputs delay	0,0040	0,0070	0,0099	0,1135	0,1967	0,2781

(b) Average correlation values for the Xoodoo implementation with registers

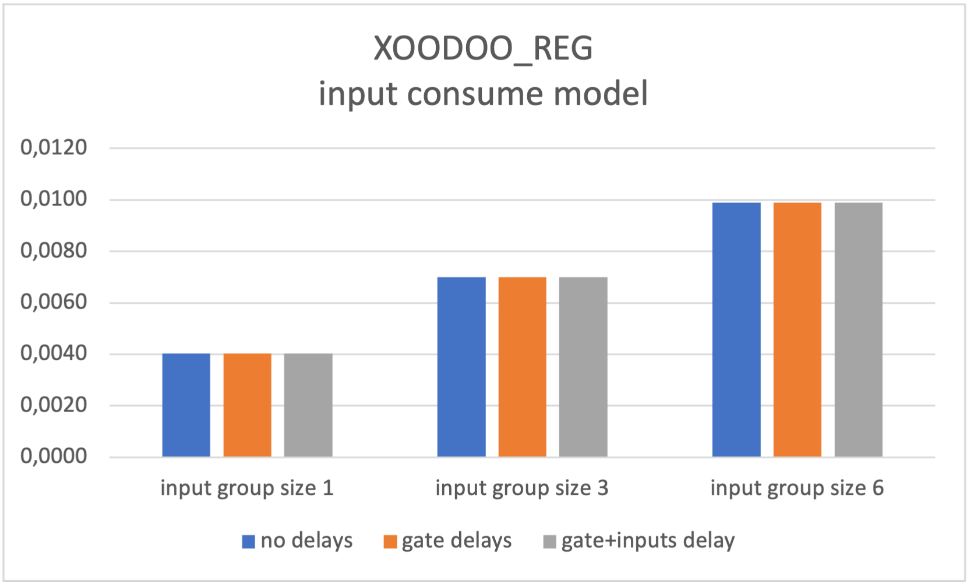
Table 5.5: Average and maximal correlation values for the Xoodoo implementation with registers

Since the outputs of this circuit are passed through a final register level, all previously introduced glitches caused by the delays are zeroed out. For this reason the correlation values don't change when adding gates' or inputs' delays.

Correlation values increase when increasing the number of bits that are grouped together for both the input and the input-state consume model. When comparing this circuit with the version without final register level, a significantly lower correlation can be noticed. The values range from 0.004 and 0.01 for the input consume model, and from 0.11 to 0.28 for the input-state consume model.



(a) Input consume model chart for the Xoodoo implementation with registers



(b) Input-state consume model chart for the Xoodoo implementation with registers

Figure 5.6: Input and input-state consume model chart for the Xoodoo implementation with registers

5.4 CMS

5.4.1 CMS without final register layer

CMS without registers Max. correlation values						
	1 bit I-CS	4 bit I-CS	8 bit I-CS	1 bit IS-CS	4 bit IS-CS	8 bit IS-CS
no delays	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
gate delays	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
gate + input delays	0,0000	0,0000	0,0000	0,4472	0,8165	1,0000

(a) Maximal correlation values for the CMS implementation without final registers

CMS without registers Avg. correlation values						
	1 bit I-CS	4 bit I-CS	8 bit I-CS	1 bit IS-CS	4 bit IS-CS	8 bit IS-CS
no delays	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
gate delays	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
gate + input delays	0,0000	0,0000	0,0000	0,1997	0,3394	0,2874

(b) Average correlation values for the CMS implementation without registers

Table 5.6: Average and maximal correlation values for the CMS implementation without registers

As can be seen from Table 5.6a, the only non-zero correlation values for this circuit are obtained, when using the input-state consume model with gate and input delays. In this case the obtained correlation values are rather high and increase, when more bits are grouped together, ranging from 0.45 to 1.

Table 5.6b shows significantly lower correlations with respect to Table 5.6a, indicating that these values vary greatly depending on the used delays and that the maximal correlation values are rarely reached.

An example of input and gate delays causing the highest correlation values are shown in tables A.6a and A.6b.

5.4.2 CMS with final register layer

When a final layer of registers is added to the previous circuit also the correlation values obtained when using the input-state consume model and with gate and input delays become zero. The resulting circuit therefore does not leak any information, even when adding delays.

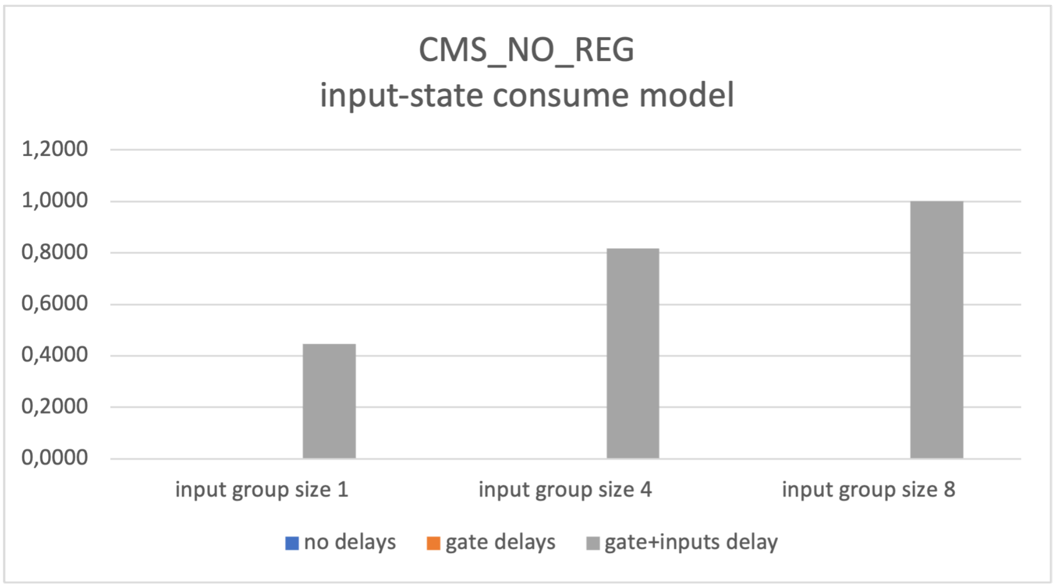


Figure 5.7: Input-state consume model chart for the CMS implementation without registers

5.5 DOM

Since this gadget can be used to protect the non linear layer of Xoodoo, we decided to test it with our tool. In particular we decided to test a 2-DOM (see Section 2.4.3 for a description of the gadget) implementation. In order to allow a better comparison with the CMS gadget, a 4-DOM was tested as well. As for Xoodoo tests were executed on a version with and on a version without a final layer of registers. On each version a first run was performed, where we calculated the correlation value for every single input bit. In the second run, the input bits were collected in groups of two, while in the third run all bits were grouped together for the calculation of the correlation.

5.5.1 2-DOM without final register layer

2-DOM without registers Max. correlation values						
	1 bit I-CS	**2 bit I-CS**	**4 bit I-CS**	**1 bit IS-CS**	**2 bit IS-CS**	**4 bit IS-CS**
no delays	0,0412	0,0509	0,0617	0,4836	0,3565	0,2315
gate delays	0,1848	0,1516	0,1109	0,8224	0,6005	0,4876
gate+inputs delay	0,2828	0,2000	0,1392	0,8224	0,7466	0,4876

(a) Maximal correlation values for the 2-DOM implementation without registers

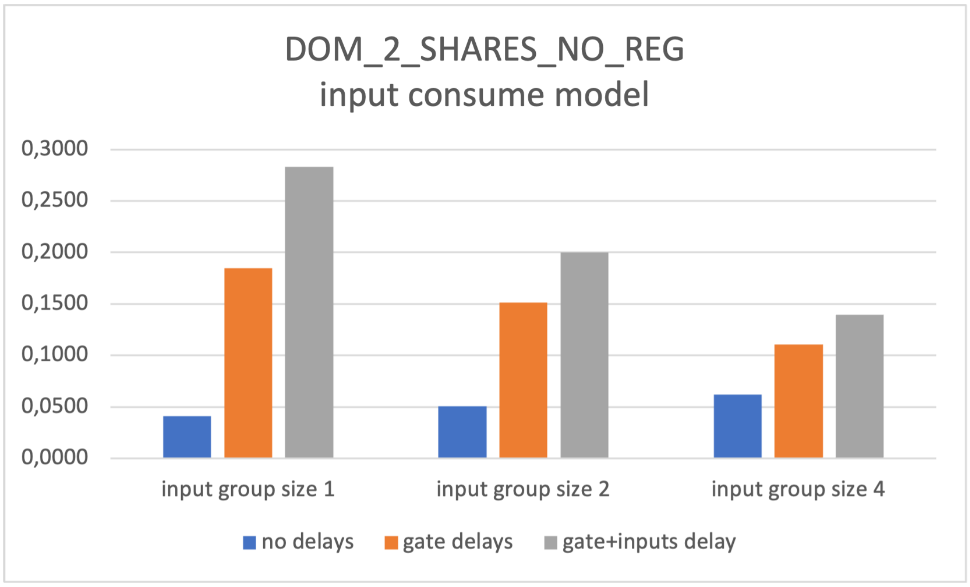
2-DOM without registers Avg. correlation values						
	1 bit I-CS	**2 bit I-CS**	**4 bit I-CS**	**1 bit IS-CS**	**2 bit IS-CS**	**4 bit IS-CS**
no delays	0,0412	0,0509	0,0617	0,4836	0,3565	0,2315
gate delays	0,1027	0,0933	0,0753	0,7166	0,5339	0,3830
gate+inputs delay	0,1046	0,1072	0,0848	0,6819	0,5349	0,3941

(b) Average correlation values for the 2 shares DOM implementation without registers

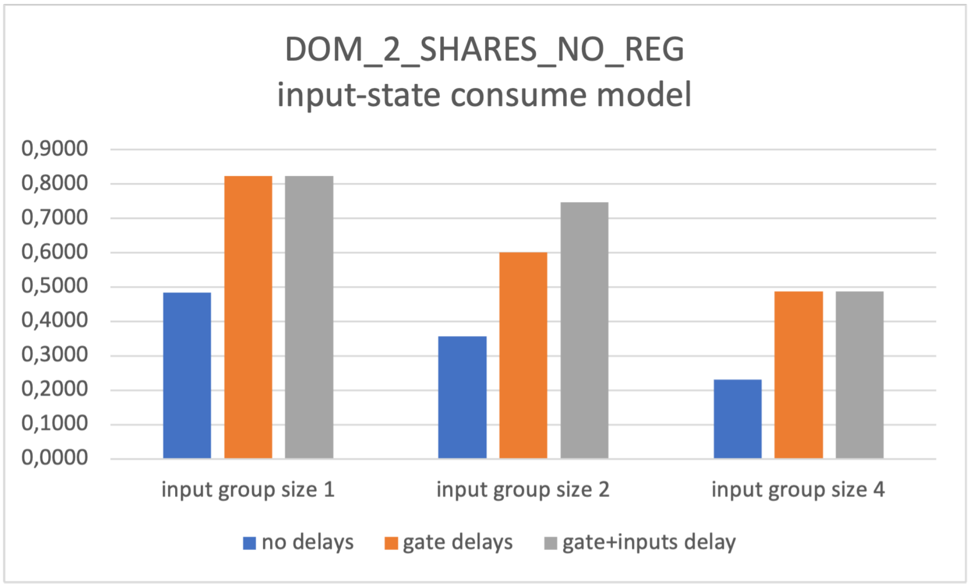
Table 5.7: Average and maximal correlation values for the 2-DOM implementation without registers

As opposed to AES-Sbox, Xoodoo and CMS the correlation for this circuit diminishes, when more bits are grouped together for both consume models. As a result the maximal correlation value is obtained, when each bit is considered separately.

Using the input-state consume model higher correlation values, which reach 0.82, are obtained, than when using the input consume model, whose correlation values max out at 0.28.



(a) Input consume model chart for the 2-DOM implementation without registers



(b) Input-state consume model chart for the 2-DOM implementation without registers

Figure 5.8: Input and input-state consume model chart for the 2-DOM implementation without registers

Table 5.7a shows how the correlation with and without input delays, when grouping 1 and 4 bits together and using the input-state consume model, are the same. This however is not true for all runs since these values have different average values, as can be seen in Table 5.7b, identifying it as a coincidence and not as a general behaviour of the circuit.

While there is not much difference between maximal and average correlation values for the input-state consume model, the average correlation values for the input consume model are just over half the size of the maximal correlation values. This indicates a greater variability of the latter, depending on the used delays.

The highest correlation is reached using the delays seen in Tables A.1a and A.1b.

5.5.2 4-DOM without final register layer

4-DOM without registers Max. correlation values						
	1 bit I-CS	**2 bit I-CS**	**4 bit I-CS**	**1 bit IS-CS**	**2 bit IS-CS**	**4 bit IS-CS**
no delays	0,0089	0,0071	0,0071	0,0071	0,0071	0,0071
gate delays	0,0067	0,4078	0,4613	0,4613	0,4613	0,4613
gate+inputs delay	0,0083	0,4110	0,5280	0,5789	0,5789	0,5789

(a) Maximal correlation values for the 4-DOM implementation without registers

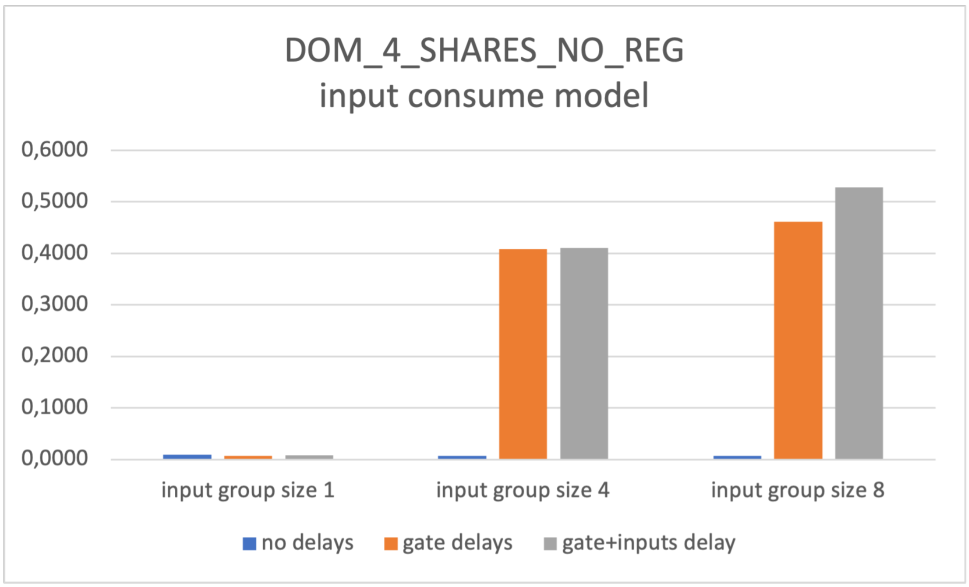
4-DOM without registers Avg. correlation values						
	1 bit I-CS	**2 bit I-CS**	**4 bit I-CS**	**1 bit IS-CS**	**2 bit IS-CS**	**4 bit IS-CS**
no delays	0,0089	0,0071	0,0071	0,0071	0,0071	0,0071
gate delays	0,0039	0,1228	0,1799	0,3213	0,3213	0,3213
gate+inputs delay	0,0039	0,1308	0,1835	0,2728	0,2988	0,3126

(b) Average correlation values for the 4-DOM implementation without registers

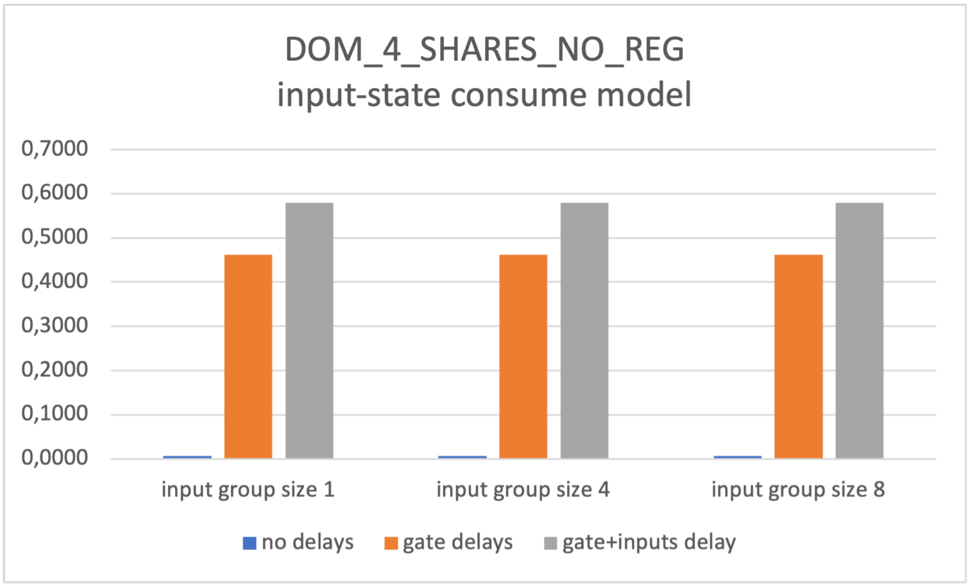
Table 5.8: Average and maximal correlation values for the 4-DOM implementation without registers

Both the input and the input-state consume model generate very low correlation values, when each input bit is considered singularly. These values considerably increase, when grouping together 4 or all 8 bits. Unlike the other circuits, the correlation values of the input and the input-state consume model reach very similar values (0.53 and 0.58 respectively).

Table 5.8a displays how the maximal correlation values of the input-state consume model don't change, when changing the number of inputs, that are grouped



(a) Input consume model chart for the 4-DOM implementation without registers



(b) Input-state consume model chart for the 4-DOM implementation without registers

Figure 5.9: Input and input-state consume model chart for the 4-DOM implementation without registers

together. Also the average value is the same for this consume model across all input group sizes, when considering no delays or when considering only gate delays. This is not true when input delays are introduced. This implies that, as long as no input delays are considered, the correlation value only depends on the chosen delays; grouping bits together does not reveal additional information.

The big difference between maximal and average values shows a great variability of the correlation values depending on the chosen delay.

The input and gate delays causing the greatest correlation for this circuit are shown in Tables A.2a and A.2b.

5.5.3 2-DOM with final register layer

2-DOM with registers Max. correlation values

	1 bit I-CS	**2 bit I-CS**	**4 bit I-CS**	**1 bit IS-CS**	**2 bit IS-CS**	**4 bit IS-CS**
no delays	0,0412	0,0509	0,0617	0,4836	0,3565	0,2315
gate delays	0,0412	0,0509	0,0617	0,4836	0,3565	0,2315
gate+inputs delay	0,0412	0,0509	0,0617	0,4836	0,3565	0,2315

(a) Maximal correlation values for the 2-DOM implementation with final registers

2-DOM with registers Avg. correlation values

	1 bit I-CS	**2 bit I-CS**	**4 bit I-CS**	**1 bit IS-CS**	**2 bit IS-CS**	**4 bit IS-CS**
no delays	0,0412	0,0509	0,0617	0,4836	0,3565	0,2315
gate delays	0,0412	0,0509	0,0617	0,4836	0,3565	0,2315
gate+inputs delay	0,0412	0,0509	0,0617	0,4836	0,3565	0,2315

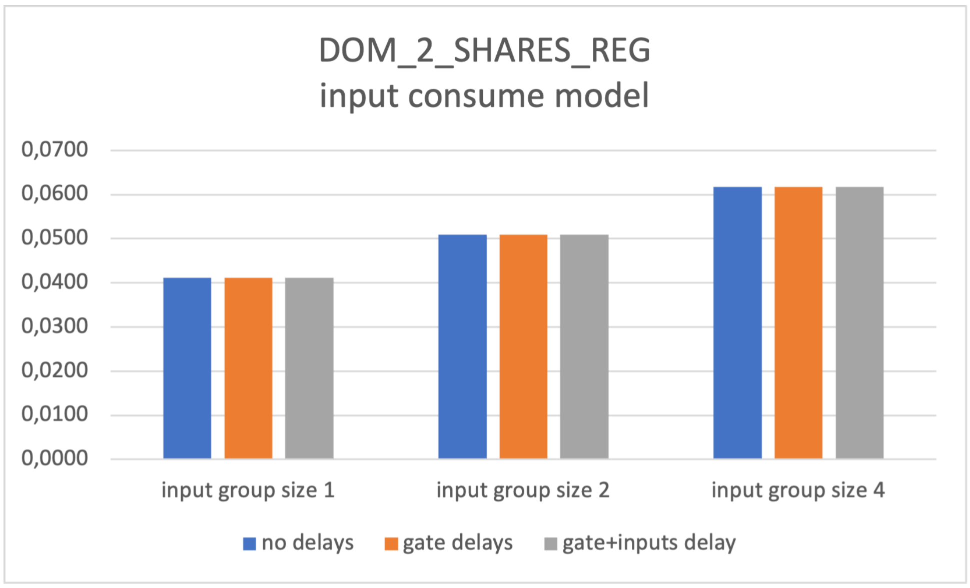
(b) Average correlation values for the 2-DOM implementation with final registers

Table 5.9: Average and maximal correlation values for the 2-DOM implementation with final registers

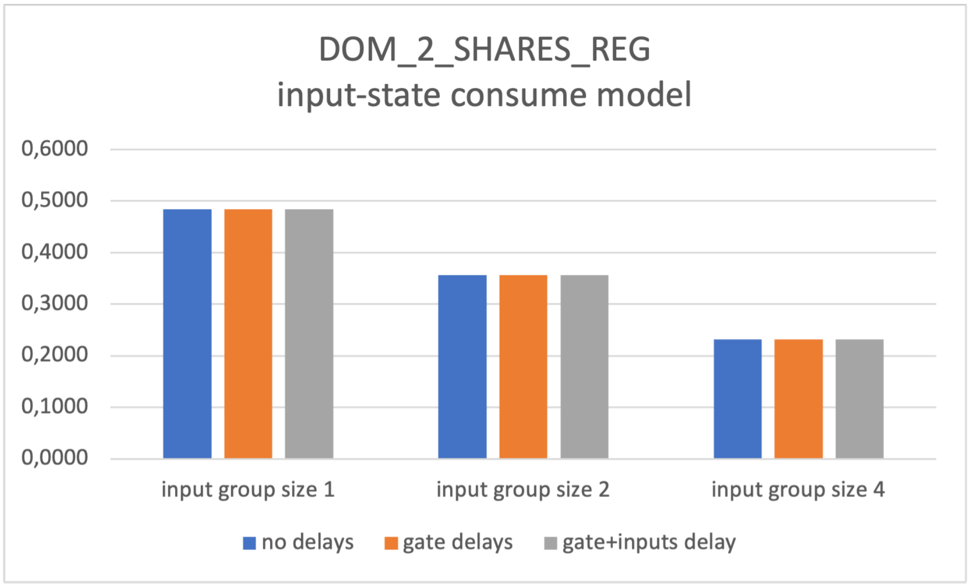
As previously seen with the CMS gadget, when a final register level is added, the correlation values stop depending on the used delays and become all equal to the correlation values of the circuit when considered without internal delays.

As for other circuits before, the correlation values of the input consume model are considerably lower than the values of the input-state consume model, with the former reaching a maximal value of 0.06 and the latter a maximal value of 0.48.

While the input consume model reaches the greatest correlation, which is 0.06, when all four input bits are grouped together, the input-state consume model reaches it's max (0.48), when every input bit is considered separately.



(a) Input consume model chart for the 2-DOM implementation with final registers



(b) Input-state consume model chart for the 2-DOM implementation with final registers

Figure 5.10: Input and input-state consume model chart for the 2-DOM implementation with final registers

5.5.4 4-DOM with final register layer

4-DOM with registers Max. correlation values						
	1 bit I-CS	**2 bit I-CS**	**4 bit I-CS**	**1 bit IS-CS**	**2 bit IS-CS**	**4 bit IS-CS**
no delays	0,0089	0,0098	0,0027	0,0034	0,0015	0,0011
gate delays	0,0089	0,0098	0,0027	0,6035	0,3007	0,3200
gate+inputs delay	0,0089	0,0098	0,0027	0,6035	0,3007	0,3200

(a) Maximal correlation values for the 4-DOM implementation with final registers

4-DOM with registers Avg. correlation values						
	1 bit I-CS	**2 bit I-CS**	**4 bit I-CS**	**1 bit IS-CS**	**2 bit IS-CS**	**4 bit IS-CS**
no delays	0,0089	0,0098	0,0027	0,0034	0,0015	0,0011
gate delays	0,0077	0,0078	0,0021	0,1284	0,0701	0,0875
gate+inputs delay	0,0077	0,0078	0,0021	0,1284	0,0701	0,0875

(b) Average correlation values for the 4-DOM implementation with final registers

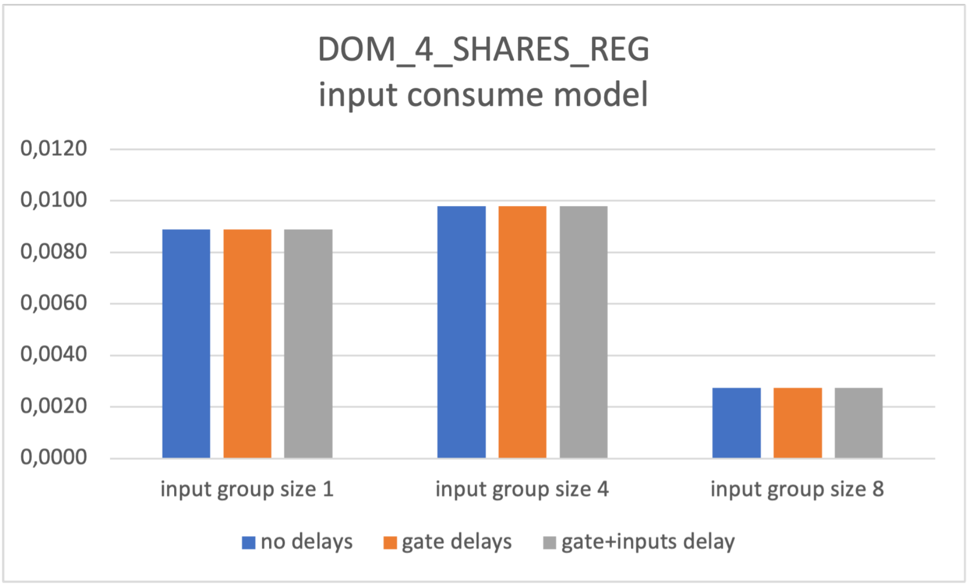
Table 5.10: Average and maximal correlation values for the 4-DOM implementation with final registers

Having a final register level, the correlation values are always the same, regardless of the chosen delays.

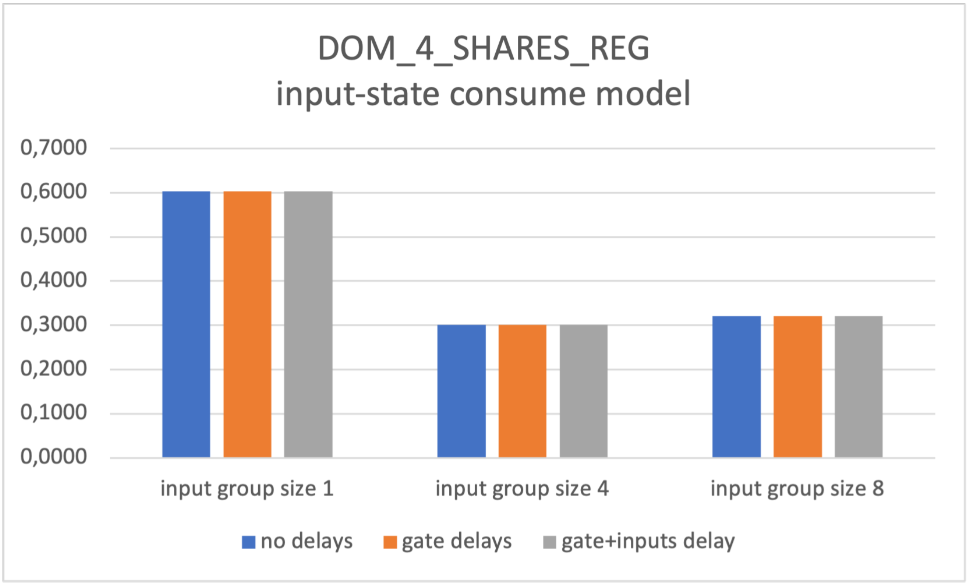
Similarly to the most part of the other gadgets, also for this circuit the correlation values reached when using the input-state consume model are considerably higher than when the input consume model is used. While the former has correlation values which reach 0.60, the greatest value of the latter is 0.10.

Even though high correlation values can be reached using the input-state consume model, the average correlation value of that model is much lower, indicating how these high values can be reached only with few delay values.

As for the 2-DOM with final register layer, the maximal correlation value is reached by considering every input bit singularly. These values then strongly decrease when more bits are grouped together.



(a) Input consume model chart for the 4-DOM implementation with final registers



(b) Input-state consume model chart for the 4-DOM implementation with final registers

Figure 5.11: Input and input-state consume model chart for the 4-DOM implementation with final registers

Chapter 6

Conclusion

The most widely used cryptographic algorithms nowadays have been studied in depth and their security can be guaranteed mathematically to a satisfactory level. When these algorithms are implemented onto physical devices in form of circuits, attackers can use so-called side channels to obtain information leaked by the circuit, which can be used to find the secret key and break the algorithm. In this thesis we develop VoLPE, a tool able to analyze the vulnerability of circuits towards Side Channel Attacks.

We put an initial effort into finding a significant model for the power consumption of a circuit during a given calculation. During computations the main power consumption of a circuit is due to the switching of the value of variables from 0 to 1 and vice versa. We decided therefore to use these switches, called toggles, as way to model the power consumption during the simulation of a circuit.

Thereafter we moved our attention towards the definition of a consume model, i.e. finding a value representing the inputs of each simulation, that could be compared to the power consumption of that simulation. The Hamming weight of the inputs has been chosen for this reason. Additionally it was deemed interesting to find a way to calculate the correlation also between the consumed power and a value representing the initial state of the circuit. For this case of interest we chose the Hamming distance between state and inputs.

After having chosen all models needed for the analysis, we proceeded to define the simulations that have to be performed on the circuits, in order to obtain relevant results. We decided to simulate each circuit under three different configurations. In a first configuration we consider an ideal circuit, which therefore ignores delays and glitch generation. In the second case gate delays are applied, which start to introduce the first glitches in the circuit; here inputs are still considered to be synchronized. Finally also input delays are considered, during the last simulation configuration.

Combining these choices we are able to develop the desired tool, able to quantify the weakness of a given circuit, simulating it. Since our analysis is based on the actual simulation of a synthesized circuit, whose structure corresponds to the physical implementation of the circuit, we are able to obtain more realistic results than tools analyzing high level descriptions of gadgets.

6.1 Future imporvements

During this work only one power consumption model has been found and implemented, alongside with only a few selection functions and consume models. In future developments of the tool, additional power consumption models, selection functions and consume models could be analysed and included, adding as well the possibility to choose the desired model before starting the computations. In the course of this thesis we focus only on modelling the power consumption, leaving therefore the opportunity to find an appropriate model for the electromagnetic fields produced by the circuit.

In the current version of VoLPE, the correlation can be calculated only between inputs and the power consumed at the outputs of the gadget. Another interesting addition to the tool could therefore be the possibility to simulate probes at internal positions of the circuit, i.e. calculating the correlation between inputs and the power consumed in a specific sub-circuit.

Furthermore it is currently not possible to simulate circuits which need a configuration phase before being able to start their computations. This is mainly due to the structure of the currently implemented testbench. In order to guarantee a wider spectrum of circuits that can be simulated with our tool, a more advanced testbench could be developed, able to also handle the initial configuration steps.

Appendix A

Tables of delay values

Input	Used delay
a0	0.48
a1	0.15
b0	0.70
b1	0.14
z0	0.83

(a) Input delays for maximal correlation values for the 2-DOM without registers

Gate	Used delay
XNOR_DELAY	0.13
NAND_DELAY	0.99
XOR_DELAY	0.27
AND_DELAY	0.05
OR_DELAY	0.04
NOR_DELAY	0.32

(b) Input delays for maximal correlation values for the 2-DOM without registers

Table A.1: Gate and input delays for maximal correlation values for the 2-DOM without registers

Input	Used delay
a0	0.16
a1	0.55
a2	0.13
a3	0.29
b0	0.76
b1	0.04
b2	0.19
b3	0.95
z0	0.39
z1	0.83
z2	0.71
z3	0.97
z4	0.60
z5	0.56

(a) Input delays for maximal correlation values for the 4-DOM without registers

Gate	Used delay
XNOR_DELAY	0.01
NAND_DELAY	0.99
XOR_DELAY	0.02
AND_DELAY	0.54
OR_DELAY	0.19
NOR_DELAY	0.75

(b) Input delays for maximal correlation values for the 4-DOM without registers

Table A.2: Gate and input delays for maximal correlation values for the 4-DOM without registers

Input	Used delay
a0	0.72
a1	0.92
a2	0.03
a3	0.27
a4	0.96
a5	0.66
a6	0.38
a7	0.31

(a) Input delays for maximal correlation values for the AES S-Box lookup table implementation

Gate	Used delay
XNOR_DELAY	0.54
NAND_DELAY	0.14
XOR_DELAY	0.37
AND_DELAY	0.26
OR_DELAY	0.00
NOR_DELAY	0.19

(b) Input delays for maximal correlation values for the AES S-Box lookup table implementation

Table A.3: Gate and input delays for maximal correlation values for the AES S-Box lookup table implementation

Input	Used delay
a0	0.60
a1	0.92
a2	0.06
a3	0.72
a4	0.55
a5	0.23
a6	0.96
a7	0.92

(a) Input delays for maximal correlation values for the AES S-Box MUX implementation performing only encryption

Gate	Used delay
XNOR_DELAY	0.19
NAND_DELAY	0.41
XOR_DELAY	0.07
AND_DELAY	0.39
OR_DELAY	0.52
NOR_DELAY	0.61

(b) Input delays for maximal correlation values for the AES S-Box MUX implementation performing only encryption

Table A.4: Gate and input delays for maximal correlation values for the AES S-Box MUX implementation performing only encryption

Input	Used delay
a0	0.36
a1	0.51
a2	0.36
a3	0.62
a4	0.76
a5	0.09
a6	0.47
a7	0.98

(a) Input delays for maximal correlation values for the AES S-Box MUX implementation performing encryption and decryption

Gate	Used delay
XNOR_DELAY	0.09
NAND_DELAY	0.31
XOR_DELAY	0.06
AND_DELAY	0.97
OR_DELAY	0.30
NOR_DELAY	0.08

(b) Input delays for maximal correlation values for the AES S-Box MUX implementation performing encryption and decryption

Table A.5: Gate and input delays for maximal correlation values for the AES S-Box MUX implementation performing encryption and decryption

Input	Used delay
a0	0.47
a1	0.90
a2	0.51
a3	0.82
b0	0.93
b1	0.76
b2	0.78
b3	0.51
r0	0.55
r1	0.59
r2	0.73
r3	0.88
r4	0.00
r5	0.26
r6	0.30
r7	0.36
r8	0.71
r9	0.31
r10	0.09
r11	0.33
r12	0.28
r13	0.03
r14	0.67
r15	0.17

(a) Input delays for maximal correlation values for the CMS implementation without registers

Gate	Used delay
XNOR_DELAY	0.03
NAND_DELAY	0.64
XOR_DELAY	0.82
AND_DELAY	0.74
OR_DELAY	0.03
NOR_DELAY	0.22

(b) Input delays for maximal correlation values for the CMS implementation without registers

Table A.6: Gate and input delays for maximal correlation values for the CMS implementation without registers

Input	Used delay
a0	0.31
a1	0.29
a2	0.03
a3	0.93
a4	0.83
a5	0.72
a6	0.35
a7	0.05
a8	0.72
a9	0.65
a10	0.68
a11	0.50

(a) Input delays for maximal correlation values for the Xoodoo implementation without registers

Gate	Used delay
XNOR_DELAY	0.04
NAND_DELAY	0.40
XOR_DELAY	0.02
AND_DELAY	0.37
OR_DELAY	0.24
NOR_DELAY	0.74

(b) Input delays for maximal correlation values for the Xoodoo implementation without registers

Table A.7: Gate and input delays for maximal correlation values for the Xoodoo implementation without registers

Bibliography

[1] Hardening macros.

[2] Ako Muhamad Abdullah et al. Advanced encryption standard (aes) algorithm to encrypt and decrypt data. *Cryptography and Network Security*, 16:1–11, 2017.

[3] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. Ironmask: Versatile verification of masking security. Cryptology ePrint Archive, Paper 2021/1671, 2021. https://eprint.iacr.org/2021/1671.

[4] Viv Bewick, Liz Cheek, and Jonathan Ball. Statistics review 7: Correlation and regression. *Critical care*, 7(6):1–9, 2003.

[5] Begül Bilgin. Threshold implementations: as countermeasure against higher-order differential power analysis. 2015.

[6] Claude Carlet. *Boolean Functions for Cryptography and Coding Theory*. Cambridge University Press, 2021.

[7] Irène Charon, Gérard Cohen, Olivier Hudry, and Antoine Lobstein. New identifying codes in the binary hamming space. *European Journal of Combinatorics*, 31(2):491–501, 2010.

[8] Ana Covic, Fatemeh Ganji, and Domenic Forte. Circuit masking: From theory to standardization, a comprehensive survey for hardware security researchers and practitioners. *arXiv preprint arXiv:2106.12714*, 2021.

[9] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Xoodoo cookbook. *Cryptology ePrint Archive*, 2018.

[10] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. The design of xoodoo and xoofff. *IACR Transactions on Symmetric Cryptology*, pages 1–38, 2018.

[11] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[12] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking aes with d+1 shares in hardware. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 194–212. Springer, 2016.

[13] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes),

2001-11-26 2001.

[14] Sebastian Faust, Vincent Grosso, SMD Pozo, Clara Paglialonga, and F-X Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. 2018.

[15] Ahmed Ghazy and Mohamed Shalan. Openlane: The open-source digital asic implementation flow. In *Proc. Workshop on Open-Source EDA Technol.(WOSET)*, 2020.

[16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *Cryptology ePrint Archive*, 2016.

[17] Owen Harrison and John Waldron. Aes encryption implementation and analysis on commodity graphics processing units. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 209–226. Springer, 2007.

[18] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Annual International Cryptology Conference*, pages 463–481. Springer, 2003.

[19] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography.* CRC press, 2020.

[20] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[21] Dwi Liestyowati. Public key cryptography. In *Journal of Physics: Conference Series*, volume 1477, page 052062. IOP Publishing, 2020.

[22] Sharad Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):950–956, 1994.

[23] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[24] Stefan Mangard, Thomas Popp, and Berndt M Gammel. Side-channel leakage of masked cmos gates. In *Cryptographers' Track at the RSA Conference*, pages 351–365. Springer, 2005.

[25] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. Investigations of power analysis attacks on smartcards. *Smartcard*, 99:151–161, 1999.

[26] L'ubica Miková, Michal Kelemen, Alexander Gmiterko, and Lukáš Kačmár. Logical circuits and their applications. *Journal of Automation and Control*, 3(3):106–109, 2015.

[27] James S Milne. Fields and galois theory (v4. 60). *order*, 3:138, 2018.

[28] Maria Chiara Molteni, Jürgen Pulkus, and Vittorio Zaccaria. On robust strong-non-interferent low-latency multiplications. *IET Information Security*, 16(2):127–132, 2022.

[29] Nicolai Müller, David Knichel, Pascal Sasdrich, and Amir Moradi. Transitional leakage in theory and practice-unveiling security flaws in masked circuits. *Cryptology ePrint Archive*, 2022.

[30] Ryan O'Donnell. *Analysis of boolean functions.* Cambridge University Press, 2014.

[31] Dag Arne Osvik, Joppe W Bos, Deian Stefan, and David Canright. Fast software aes encryption. In *International Workshop on Fast Software Encryption*, pages 75–93. Springer, 2010.

[32] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Annual Cryptology Conference*, pages 764–783. Springer, 2015.

[33] Matthew JB Robshaw. Stream ciphers. *RSA Labratories*, 25, 1995.

[34] Gustavus J Simmons. Symmetric and asymmetric encryption. *ACM Computing Surveys (CSUR)*, 11(4):305–330, 1979.