Masters' Internship Report

Sorbonne Université

CryptoExperts

Discipline : Sécurité, Fiabilité et Performance du Numérique (SFPN)

VeriSiCC: Verification of Side-Channel Countermeasures

By : **Abdul Rahman TALEB**

Supervisors : Matthieu RIVAIN
Sonia BELAID

Date : August 24, 2020

Acknowledgements

First of all, I would like to thank my supervisors Matthieu RIVAIN and Sonia BELAÏD, whose expertise was invaluable in the formulating of the research topic and methodology in particular. They made my internship a very unique experience, which exceeded all of my expectations. They never hesitated to help me when in difficulty, to answer my questions, and to be ready to take of their time for brainstorming sessions when needed. And for that, I am extremely grateful.

I would also like to thank Professor Damien VERGNAUD, who was the main reason I found this opportunity. He was my internship supervisor in my first year of Masters, and he was the one who recommended me to CryptoExperts company. Without him, I would never have been where I am now, and I would not have found the phD offer at the company. And I am honored to have him again as my thesis director for my upcoming phD.

Finally, I would like to thank all of the teachers I had during my studies, who helped get to where I currently am, including my Masters' degree supervisors Professor Mohab SAFEY EL DIN, and especially professor Valérie MENISSIER-MORAIN, who also helped me make my choice of specialty for my Masters' studies.

Abstract

Side-channel attacks are considered a major risk against cryptographic implementations. Since their introduction in the nineties, researchers have been trying relentlessly to protect against such attacks. Higher-order masking is among the most powerful countermeasures to counteract side-channel attacks. Meanwhile, trying to prove the efficiency of masking is not an easy job, since one needs to formalize the leakage model on devices and theoretically exhibit the protection of such a countermeasure. Until now, the most widely used one in the community is the probing leakage model. While this model is convenient for security proofs, it has recently been challenged since it does not capture an adversary exploiting full leakage traces as, e.g., in horizontal attacks.

The random probing leakage model is one of the closest models to reality. In this model, each variable is leaked with a certain probability depending of its number of uses within the implementation. However, proving security in this model is more challenging than in the probing model, and until today, not a lot of contributions have been made.

In this report, I provide a detailed overview of the work accomplished during my Masters' internship at CryptoExperts, which was in continuity to an already existing research project. The work defines a framework for the random probing model. The goal of my internship was to implement the automatic tool **VRAPS** which quantifies the random probing security of an algorithm from its leakage probability. We also formalize a composition property for secure random probing gadgets, and an expansion strategy for circuits achieving an arbitrary level of security using a random probing expandability property. I implement this expansion strategy and use it for a concrete implementation of the AES algorithm.

As the expansion strategy is inspired from the work by Ananth, Ishai, and Sahai (AIS) (CRYPTO 2018), I also provide a comparison between our strategy and theirs, in terms of properties, complexity and tolerated leakage probability. We provide an instantiation for our strategy with gadgets that tolerate a leakage probability up to 2^{-8}, against 2^{-26} for AIS instantiation, with a better asymptotic complexity.

Contents

Notations

- We note $x \xleftarrow{\$} S$ for a variable x which value is assigned uniformly at random from the set of values S.
- We note $[n], n \in \mathbb{N}$ to be the set of integers $[1, \ldots, n]$.
- We shall denote by $\mathbf{SD}(D_1; D_2)$ the statistical distance between the probability distributions D_1 and D_2. We have :
$$\mathbf{SD}(D_1; D_2) = \frac{1}{2} \sum_x |p_{D_1}(x) - p_{D_2}(x)|$$
where p_{D_1} and p_{D_2} denote the probability mass functions respectively for D_1 and D_2.
- We say that two probability distributions D_1 and D_2 are ϵ-***close*** if $\mathbf{SD}(D_1; D_2) \leq \epsilon$. We denote it $\mathbf{D_1} \approx_\epsilon \mathbf{D_2}$. $\mathbf{D_1} \overset{\mathbf{id}}{=} \mathbf{D_2}$ means that $\mathbf{D_1}$ and $\mathbf{D_2}$ are identically distributed ($\epsilon = 0$).

Chapter 1

Introduction

This chapter is intended to give an introduction to the internship's subject, missions and contributions. I will first start by giving a brief overview of the company Cryptoexperts. Then, I will describe the main problematic that was studied and the contributions.

1.1 Description of the Company

CryptoExperts

Address : 41 Boulevard des Capucines, 75002 Paris, France.
Mail : contact@cryptoexperts.com
Website : https://www.cryptoexperts.com/
Team :

- Matthieu Rivain, **CEO and Senior Cryptography Expert**
- Pascal Paillier, **Senior Cryptography Expert**
- Sonia Belaïd, **Cryptography Expert**
- Aleksei Udovenko, **Cryptography Expert**
- Junwei Wang, **PhD Student**
- Louis Goubin, **Scientific Advisor**
- Antoine Joux, **Scientific Advisor**

CryptoExperts SAS is an IT security and cryptography startup company founded in 2009. Its main mission is to fill the gap between the scientific state of the art in cryptography and the technology

found in current security products. The team is made of internationally recognized industrial and academic researchers in security and cryptography, who are able to provide consulting services, and lead research projects in their R&D department. Two students and I have joined the team at the same time for 6 months internship positions, each working on a different subject.

1.2 Problematic & State of the Art

Side-Channel Attacks. Nowadays, cryptography is omnipresent on various embedded systems such as smart cards. The encryption algorithms on these components have to provide high security against malicious actions. Cryptosystems are often proven to be secure theoretically, assuming that the adversary is limited to the observation of the inputs and outputs, e.g., in the so-called **black-box** model. However, this model does not faithfully reflect the reality of embedded devices. In fact, a more realistic model was introduced in the nineties [18], in which the attacker can observe the physical leakage of the running device. These attacks are referred to as side-channel attacks (or SCA). They operate in a **grey-box model** and often exploit the dependence between sensitive values of an algorithm and the physical leakage of the device (execution time, power consumption, electromagnetic radiation, ...). Since these attacks only require cheap equipment and can be mounted efficiently, a lot of studies have been going on to find effective countermeasures to secure implementations in the grey-box model. The most deployed countermeasure against side-channel attacks is currently what we call *higher-order masking*.

Higher-order masking. The masking countermeasure was originally introduced by Chari et al [11] and by Goubin and Patarin [16]. The idea is to split the value of each sensitive information x into n shares $x_0, \ldots, x_{n-1}$ that satisfy for a certain operation $\star$ associated to a group G:

$$x = x_0 \star \cdots \star x_{n-1}$$

where $x_1 \xleftarrow{\$} G, \ldots, x_{n-1} \xleftarrow{\$} G$, and $x_0 = x \star^{-1} x_1 \star^{-1} \cdots \star^{-1} x_{n-1}$. When $\star = \oplus$ (bitwise XOR), the masking is said to be *Boolean*. Given this scheme, the knowledge of at most $n-1$ shares does not reveal any information on the sensitive variable and an attacker has to combine the leakage of n variables to recover the sensitive data. As leakage comes with noise, the combination of shares has been shown to be exponential in the number of data to combine, that is in the masking order n [11]. When higher-order masking is used to design small cryptographic functions, one should be able to prove the security of such functions in theory.

Formal verification of security against SCA. In general, to define a security notion when using masked implementations (that we call circuits), *leakage models* were introduced. There are three main models that vary in terms of convenience for security proofs and closeness to reality of physical leakage.

The most popular one in the community is the probing model [17], where a circuit is considered t-probing secure if any set of t intermediate variables do not reveal any information on the sensitive secret variables. The motivation behind this model is that as t grows, recovering t variables from measurements becomes more difficult and complex, since these measurements reveal noisy functions of the variables. Several works have studied the security of masked implementations in this model [8, 21, 13]. Additionally, formal tools have been constructed to automatically verify the security and detect flaws in masked circuits in this model [4, 10, 12]. Compilers have also been proposed to generate t-probing secure masked implementations for any t, given the description of a primitive [5].

Meanwhile, the probing model has recently been challenged as it does not truthfully capture the reality of embedded devices. One of its main flaws is that it does not take into account the fact that some variables can be observed several times, and thus be revealed with a lower level of noise, like for example in horizontal attacks [7].

A model that better captures the reality of leakage is the random probing model, where each variable leaks with some fixed probability p. A circuit is then secure if the probability that the leaking wires reveal any information about the secrets is negligible. This model obviously considers the case of repeated manipulations of intermediate variables, and thus better captures attacks like the stated horizontal attacks [7]. However, proving security in this model is more complex than in the probing model.

An even more realistic model is the noisy leakage model introduced in [11] and [20]. In this model, a circuit is considered secure if it is difficult for an attacker to recover the secrets from the noisy functions of intermediate variables. While it very well approaches a real life scenario, establishing security proofs in this model is not convenient.

Until recently, security continued to be proved in the probing model. In addition, a reduction property was introduced in [15] which allows to reduce proofs from the probing model to the noisy leakage model, while passing through the random probing model :

$$\textbf{Probing Model} \implies \textbf{Random Probing Model} \implies \textbf{Noisy Leakage Model}$$

In a nutshell, if a circuit C of size $|C|$ is t-probing secure, then C is also $\mathcal{O}(t/|C|)$-random probing secure. And if a circuit is p-random probing secure, then it is also $\mathcal{O}(p)$-noisy leakage secure. It is obvious that the reduction from probing to random probing is not very *tight*, since the security

parameter is reduced as the size of the circuit grows. Meanwhile, the reduction from random probing to noisy leakage is tighter since it only replaces the security parameter p by $\mathcal{O}(p)$. For this reason, many works (including ours) are recently trying to establish security properties in the random probing model, since it is still more convenient than the noisy leakage model, and the reduction to noisy leakage (which is the most realistic model) is convenient.

Constructions in the random probing model. Some constructions that were proven secure in the probing model have also been proven secure in the random probing model, but the tolerated leakage probability was not constant, contrarily to what is needed in practice.

A few constructions actually tolerate a constant leakage probability [1, 2, 3]. The two works [1] and [3] are based on expander graphs, but the leakage probability is not made explicit. The third work [2] is based on secure multi-party computation protocols and an expansion strategy. The latter offers an instantiation that tolerates a probability of around 2^{-26} for a circuit of size $|C|$ gates. Meanwhile, the complexity expresses as $\mathcal{O}(|C|.poly(\kappa))$ for a security parameter κ, with a polynomial that is not made explicit.

1.3 Contributions

When I started my internship, my supervisors have already been working on a framework to verify, compose and build random probing secure circuits from simple gadgets. They were writing a paper stating some of the results they got. In the continuity of their work, we were able to complete the paper ***Random Probing Security: Verification, Composition, Expansion and New Constructions***, *by Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain and Abdul Rahman Taleb*, published at *CRYPTO 2020* [9]. Mainly, we were able to accomplish the following contributions :

Automatic Verification Tool. We exhibit a formal verification method to automatically verify the random probing security of small circuits, composed of addition and multiplication operations, for a certain leakage probability p. In a nutshell, a circuit is (p, f)-random probing secure if it leaks information on the secret with probability $f(p)$, where $f(p)$ is the failure probability function. Based on this method, we implemented a tool named **VRAPS** ((V)erifier of (RA)ndom (P)robing (S)ecurity) based on top of a set of rules that were previously defined to verify the probing security of implementations [4]. The tool takes as input the description of a circuit and outputs upper and lower bounds on the failure probability function. While the verification complexity is limited to small circuits, the state-of-the-art shows that verifying those circuits can be useful in practice

(see e.g. the **maskVerif** tool [4]), for instance to verify gadgets and then deduce global security through composition properties and/or low-order masked implementations. While my supervisors have already been working on an implementation of the tool, I spent a considerable part of my internship trying to complete the implementation, and optimize the code to increase the size of the circuits that the tool can verify. The final source code of **VRAPS** is publicly available [1].

Composition and Expanding Compiler. Before I started my internship, the rest of the authors of the paper were able to introduce a composition security property (RPC for Random Probing Composition) to make gadgets composable in a global random probing secure circuit. They additionally introduced a property of random probing expandability (RPE), inspired from the secure multi-party computation protocol based approach from [2], which allows a circuit to achieve arbitrary levels of random probing security. On my arrival, we studied further this modular approach [2] and compared it with our construction of an expanding compiler that builds random probing secure circuits from small base gadgets. We also integrated the verification of RPC and RPE properties in the implemented tool **VRAPS**. Additonally, we provide an implementation of the expanding compiler in python, as well as an implementation of the AES128 algorithm[2] using expanded random probing secure circuits for operations, and masked variables.

Instantiation. We instantiate our expanding compiler with a construction of small gadgets that satisfy the RPE property under the correct conditions. For a security level κ, we show using our tool **VRAPS** that our instantiation achieves a complexity of $\mathcal{O}(\kappa^{7.5})$ and tolerates a constant leakage probability $p \approx 0.0045 > 2^{-8}$. In comparison, we provide a precise analysis of the construction from [2] and show that it achieves an $\mathcal{O}(\kappa^{8.2})$ complexity for a much lower tolerated leakage probability ($p \approx 2^{-26}$).

Further Analysis. In the continuity of the above results, we started studying the aspects of complexity and tolerated leakage probability in the random probing model, and we tried to formalize some useful bounds, and obtain some results that might open doors for future works. These contributions will follow on in a phD position that was offered to me at the company, and that will begin on the first of October 2020.

[1] **https://github.com/CryptoExperts/VRAPS**
[2] **https://github.com/CryptoExperts/poc-expanding-compiler**

Chapter 2

Random Probing Model

In this chapter, I will formally define the notions and terms related to the random probing model. I will define the random probing security of a circuit, and the composition property that allows to compose small random probing secure circuits, into a global random probing secure circuit under certain conditions. I will also introduce the expansion strategy that allows a circuit composed of smaller secure circuits, to achieve an arbitrary level of security. This chapter summarizes all of the theoretical results of our paper [9] that were established **before** and **during** my internship (before my internship, the co-authors of the paper have already been working on the project). Due to lack of space, for proofs of the results, we refer you to our recent publication [9].

2.1 Preliminaries

I will define in this section the basic notions related to circuits and circuit compilers used for our work.

Definition 1 (Randomized Arithmetic Circuit). An l-to-m randomized arithmetic circuit is a labeled directed acyclic graph whose edges are wires (variables) and vertices are arithmetic gates processing operations over $\mathbb{K}$. It has l inputs, m outputs and is composed of the following gates:

- 2-to-1 addition gates for addition operations over $\mathbb{K}$.
- 2-to-1 multiplication gates for multiplication operations over $\mathbb{K}$.
- 1-to-2 copy gates to output two independent fresh copies of their input.
- 0-to-1 random gates to generate a uniform random value over $\mathbb{K}$.

If the circuit does not contain random gates, then it is called an l-to-m **arithmetic circuit**.

Definition 2 (Circuit Compiler (General Definition)). A circuit compiler is a set of algorithms (CC, Enc, Dec) defined over a field $\mathbb{K}$:

- CC is a deterministic circuit compilation algorithm that takes as input an arithmetic circuit C and outputs a randomized arithmetic circuit $\hat{C}$.
- Enc is a probabilistic encoding algorithm that takes the circuit input $x \in \mathbb{K}^l$, and encodes it into $\hat{x} \in \mathbb{K}^{l'}$ for some $l' \in \mathbb{N}$.
- Dec is a deterministic decoding algorithm that takes the encoded output $\hat{y} \in \mathbb{K}^{m'}$ for some $m' \in \mathbb{N}$, and decodes it into $y \in \mathbb{K}^m$.

such that these algorithms verify:

- **Correctness:** $Dec(\hat{C}(\hat{x})) = C(x)$, $\forall x \in \mathbb{K}^l$, for any arithmetic circuit C.
- **Efficiency:** For a security parameter $\lambda \in \mathbb{N}$, the running time of CC is polynomial in $|C|$ (number of gates in C) and in λ. Also, the running time of Enc (resp. Dec) is polynomial in $|x|$ (resp. $|\hat{y}|$) and in λ.

For our work, we mainly focus on n-linear sharing for the encoding and decoding algorithms. That is given a variable $x = (x_1, \ldots, x_l) \in \mathbb{K}^l$, the linear encoding probabilistic algorithm denoted $LinEnc$ produces $x' = (x'_1, \ldots, x'_l) \in (\mathbb{K}^n)^l$, where each $x'_i \in \mathbb{K}^n$ is a n-sharing of x_i. In other terms, $x'_i = (x'_{i,1}, \ldots, x'_{i,n})$, where $x'_{i,1} \xleftarrow{\$} \mathbb{K}, \ldots, x'_{i,n-1} \xleftarrow{\$} \mathbb{K}$ and $x'_{i,n} = x'_i + x'_{i,1} + \cdots + x'_{i,n-1}$.
The decoding n-linear sharing deterministic algorithm denoted $LinDec$ takes as input a n-sharing variable $x' = (x'_1, \ldots, x'_l) \in (\mathbb{K}^n)^l$, and outputs $x = (x_1, \ldots, x_l) \in \mathbb{K}^l$ such that $x_i = x'_{i,1} + \cdots + x'_{i,n}$.

Definition 3 (n-share l-to$-m$ gadget). Let $g : \mathbb{K}^l \to \mathbb{K}^m$ a function. An n-share l-to-m gadget is a randomized arithmetic circuit that maps an input $x' \in (\mathbb{K}^n)^l$ to an output $y' \in (\mathbb{K}^n)^l$ such that $x = LinDec(x') \in \mathbb{K}^l$ and $y = LinDec(y') \in \mathbb{K}^m$ and $y = g(x)$.

We finally define the circuit compiler that we use for the rest of our definitions.

Definition 4 (Standard Circuit Compiler). Let $\lambda \in \mathbb{N}$ be a security parameter and let $n = poly(\lambda)$. Let G_{add}, G_{copy}, G_{mult} be n-share gadgets respectively for functions addition, copy and multiplication over $\mathbb{K}$. A standard circuit compiler with sharing order n and base gadgets G_{add}, G_{copy}, G_{mult} is the circuit compiler (CC, Enc, Dec) satisfying the following:

- Enc is n-sharing $LinEnc$.

- Dec is n-sharing $LinDec$.
- The circuit compilation CC consists in replacing each gate in the original circuit by an n-share gadget with corresponding functionality (either G_{add}, G_{copy} or G_{mult}), and each wire by a set of n wires carrying a n-linear sharing of the original wire. If the input circuit is a randomized arithmetic circuit, each of its random gates is replaced by n random gates, which duly produces a n-linear sharing of a random value.

For such a circuit compiler, the correctness and efficiency directly holds from the correctness and efficiency of the gadgets G_{add}, G_{copy} and G_{mult}.

Remark 1. As explained in the above definition, to compile a random gate, the latter is replaced by n random gates, since an n-linear sharing of a random value is simply n independent random values. This procedure is the definition of what an n-share random gadget is.

2.2 Random Probing Leakage

Let $p \in [0, 1]$ be a fixed probability parameter. In the random probing model, we consider that each variable in a circuit leaks its value independently with leakage probability p. The definition of security in the random probing model relies on two probabilistic algorithms which operate on a circuit C.

- **LeakingWires**$(C,\ p)$. This procedure takes as input a circuit C, the leakage probability p, and outputs a set W of wire labels that includes each wire label from C with probability p. This set W can be thought of as a random variable output from a random process in which each wire in the circuit is added to W with probability p independently of the other wires. We consider that the leaking wires are only the input and intermediate wires of the circuit, and the output wires are not included in the leakage. This is mainly because when composing several circuits (or gadgets), the output wires of a circuit are the input wires of another circuit. So leakage on output wires can be considered in the input wires of another circuit.
- **AssignWires**$(C,\ W,\ x)$. This procedure takes as input a circuit C, a set of wire labels $W = \textbf{LeakingWires}(C, p)$, and an input x. The procedure then runs the circuit C on input x and outputs a $|W|$-tuple $w \in (\mathbb{K} \cup \perp)^{|W|}$ that corresponds to the assigned values to each wire labeled in W during the execution.

Definition 5 (Random Probing Leakage). The p-random probing leakage of a randomized arithmetic circuit C on input x is the distribution $\mathcal{L}_p(C, x)$ obtained as

$$\mathcal{L}_p(C, x) \stackrel{\text{id}}{=} \textbf{AssignWires}(C, \textbf{LeakingWires}(C, p), x) \ .$$

In general, a circuit C is considered secure in the random probing model if the corresponding random probing leakage does not reveal any information about the secret values. To formally define this, we use the notion of simulator to say that if there exists a simulator algorithm that has no information on the secret, and that can output a distribution identically distributed to $\mathcal{L}_p(C, x)$, then this means that the values leaked during the execution do not directly depend on the secret inputs of the circuit.

Definition 6 (Random Probing Security). A n-share randomized arithmetic circuit C with $l.n \in \mathbb{N}$ input gates is (p, ϵ)-random probing secure with respect to encoding Enc if there exists a simulator Sim such that for every $x \in \mathbb{K}^l$:

$$out \leftarrow Sim(C, \textbf{LeakingWires}(C, p)) \ .$$

such that

$$\Pr[out = \perp] = \epsilon \quad \text{and} \quad \Big(out \mid out \neq \perp\Big) \stackrel{\text{id}}{=} \Big(\mathcal{L}_p(C, Enc(x)) \mid out \neq \perp\Big) \ .$$

A circuit compiler (CC, Enc, Dec) is (p, ϵ)-random probing secure if for every (randomized) arithmetic circuit C the compiled circuit $\hat{C} = CC(C)$ is $(p, |C|.\epsilon)$-random probing secure where $|C|$ is the size of the original circuit.

This procedure is called a Simulation with abort, where ϵ is called the probability of **Simulation Failure**. This definition can be equivalently seen as the case where the simulation is without an abort and where $out \approx_\epsilon \mathcal{L}_p(C, x)$.
A simulator aborts on an acquired set W if the simulation of all wires of C labeled in W requires the full knowledge of any of the l values in the secret input $x \in \mathbb{K}^l$. In other terms, at least one n-sharing of the l input sharings in $Enc(x) = (x'_1, \ldots, x'_n) \in (\mathbb{K}^n)^l$ enters in the computation of the wires' values in W. Then a circuit is said to be (p, ϵ)-random probing secure if the simulation abort probability is bounded by ϵ.

2.2.1 Simulation Failure Probability

Consider a circuit C with a total of s wires labeled from 1 to s. Since each wire in the circuit leaks independently with probability p, then it can be observed that when acquiring a set of wires W from the **LeakingWires(C, p)** procedure, the probability of having such a set W follows a binomial distribution, with respect to the leakage probability p and the total number of wires s. That is

$$Pr(W) = p^{|W|}.(1-p)^{s-|W|}$$

For each such set $W \subseteq [s]$, the simulator either aborts and outputs $\perp$, or outputs a distribution that is identical to $\mathcal{L}_p(C, x)$ of the random probing leakage.
Since the simulation abort occurs with probability ϵ, then ϵ can be expressed as a function $f(p)$ of p:

$$\epsilon = f(p) = \sum_{\substack{W \subseteq [s] \\ \text{Simulation Failure} \\ \text{on W}}} p^{|W|}(1-p)^{s-|W|}$$

In other words, ϵ is the sum of the probabilities of acquiring a set $W \subseteq [s]$ for which there is a simulation failure (W requires the full knowledge of at least one of the input sharings to be simulated).
If we denote $\mathbf{c_i}$ to be the number of sets W such that $|W| = i$ and there is a simulation failure on W, then ϵ can be equivalently expressed as

$$\epsilon = f(p) = \sum_{i=1}^{s} \mathbf{c_i}\ p^i(1-p)^{s-i}$$

It is easy to see that an upper bound can be derived on ϵ from the fact that $\mathbf{c_i} \leq \binom{s}{i}$, so

$$f(p) \leq \sum_{i=1}^{s} \binom{s}{i}\ p^i(1-p)^{s-i}$$

This expression of ϵ is going to be used as a base for our automatic verification tool of random probing security which will be exhibited in the next chapter. The tool called **VRAPS** takes as input a circuit C with s wires and computes bounds on the function $\epsilon = f(p)$ for which the circuit is (p, ϵ)-random probing secure. It also estimates upper and lower bounds on the tolerated leakage probability p by the circuit.

2.3 Random Probing Composability

When using standard circuit compilers of sharing order n and base gadgets G_{add}, G_{copy} and G_{mult}, one would like to show that compiling a circuit C by replacing each basic gate by the corresponding gadget and each wire by n wires, outputs a compiled circuit that is globally random probing secure. The goal of the composition notion is to ensure that if the building blocks gadgets are random probing composable under certain conditions, then a circuit that composes them will be random probing secure. The main result of this property is:

$$\begin{array}{c} G_{\text{add}},\, G_{\text{copy}} \text{ and } G_{\text{mult}} \text{ are} \\ (t,p,\epsilon)\text{-Random probing composable} \end{array} \quad \Longrightarrow \quad \begin{array}{c} \text{Compiled circuit } \hat{C} \text{ is} \\ (p, |C|.\epsilon)\text{-Random probing secure} \end{array}$$

for certain parameters ϵ and t.
Let us give the definition of random probing composability that allows to satisfy the above reduction. In the following definition, for an n-share, l-to-m gadget, we denote by I a collection of sets $I = (I_1,\ \ldots,\ I_l)$ with $I_1 \subseteq [n], \ldots, I_l \subseteq [n]$ where $n \in \mathbb{N}$ refers to the number of shares. For some $x' = (x'_1,\ \ldots,\ x'_l) \in (\mathbb{K}^n)^l$, we then denote $x'|_I = (x'_1|_{I_1},\ \ldots,\ x'_l|_{I_l})$ where $x'_i|_{I_i} \in \mathbb{K}^{|I_i|}$ is the tuple composed of the coordinates of the sharing x'_i of indexes included in I_i.

Definition 7 (Random Probing Composability). Let $n, l, m \in \mathbb{N}$. An n-share gadget $G : (\mathbb{K}^n)^l \to (\mathbb{K}^n)^m$ is (t,p,ϵ)-random probing composable (RPC) for some $1 \leq t < n$ and $p, \epsilon \in [0,1]$, if there exists a deterministic algorithm Sim_1^G and a probabilistic algorithm Sim_2^G such that for every input $x' \in (\mathbb{K}^n)^l$ and for every set collection $J_1 \subseteq [n], \ldots, J_m \subseteq [n]$ of cardinals $|J_1| \leq t$, $\ldots$, $|J_m| \leq t$, the random experiment

$$\begin{aligned} &W \leftarrow LeakingWires(G,p) \\ &I \leftarrow Sim_1^G(W,J) \\ &out \leftarrow Sim_2^G\Big(x'|_I\Big) \end{aligned}$$

yields

$$\Pr\Big((|I_1| > t) \vee \ldots \vee (|I_l| > t)\Big) \leq \epsilon \tag{2.1}$$

and

$$out \overset{\text{id}}{=} \Big(AssignWires(G,W,x')\ ,\ y'|_J\Big)$$

where $J = (J_1, \ldots, J_m)$ and $y' = G(x')$. Let $f : \mathbb{R} \to \mathbb{R}$. The gadget G is (t,f)-RPC if it is

$(t, p, f(p))$-RPC for every $p \in [0, 1]$.

Informally, the definition of random probing composability also uses the notion of simulator as in definition 6. This time, the simulator will first acquire a set of leaking wires W using the $LeakingWires$ procedure. It will also acquire any set of indices of shares $J = (J_1, \ \ldots, \ J_m)$ of sizes at most t for each of the output sharings in $y' = (y'_1, \ \ldots, \ y'_m)$. Then in the first part $(Sim_1^G(W, J))$, the simulator will determine the set of input shares $I = (I_1, \ \ldots, \ I_l)$ of inputs $x' = (x'_1, \ \ldots, \ x_l)$ necessary for a perfect simulation of W and J. Note that there always exists such a collection of sets I since $I = ([n], \ \ldots, \ [n])$ trivially allows a perfect simulation whatever W and J.
However, the goal of Sim_1^G is to return a collection of sets I with cardinals at most t. Thus, if the simulation needs more than t shares of any of the input sharings, then the simulation fails. Otherwise, the second part of the simulator (Sim_2^G) will generate uniformly at random the shares $x'|_I$ and then run the gadget G to obtain the wire values indexed in W and J. This produces a perfect simulation of W and J giving $out \overset{\text{id}}{=} \left(AssignWires(G, W, x') \ , \ y'|_J\right)$.
The idea behind this constraint is to keep the following composition invariant: for each gadget we can achieve a perfect simulation of the leaking wires plus t shares of each output sharing from t shares of each input sharing. We shall call **failure event** the event that at least one of the sets $I_1, \ \ldots, \ I_l$ output of Sim_1^G has cardinality greater than t. When (t, p, ϵ)-RPC is achieved, the failure event probability is upper bounded by ϵ according to (2.1). A failure event occurs whenever Sim_2^G requires more than t shares of one input sharing to be able to produce a perfect simulation of the leaking wires (i.e. the wires with label in W) together with the output shares in $y'|_J$. Whenever such a failure occurs, the composition invariant is broken. In the absence of failure event, the RPC notion implies that a perfect simulation can be achieved for the full circuit composed of RPC gadgets. This is formally stated in the next theorem, giving us the desired reduction property stated at the beginning of this section. The full proof of this theorem is in our published paper [9].

Theorem 1 (Composition Security). Let $t \in \mathbb{N}$ such that $1 \leq t < n$ for a sharing order n, $p, \epsilon \in [0, 1]$, and CC be a standard circuit compiler with (t, p, ϵ)-RPC base gadgets. For every (randomized) arithmetic circuit C composed of $|C|$ gates, the compiled circuit $\hat{C} = CC(C)$ is $(p, |C|.\epsilon)$-random probing secure. Equivalently, the standard circuit compiler CC is (p, ϵ)-random probing secure.

2.3.1 Simulation Failure Probability

Recall that for (p, ϵ)-random probing security, ϵ is expressed as

$$f(p) = \sum_{i=1}^{s} c_i \; p^i (1-p)^{s-i}$$

with c_i being the number of sets W such that $|W| = i$ and the simulation of W requires n shares of any of the input sharings.
In the case of the composability property, the simulation also includes a collection of sets $J = (J_1, \ \ldots, \ J_m)$ of t output wires indices for each output sharing. Then, ϵ in (t, p, ϵ)-random probing composability can be redefined as

$$\epsilon = f(p) = \sum_{i=1}^{s} \; \max_{J} \mathbf{c_i^J} \;\; p^i (1-p)^{s-i}$$

where $\mathbf{c_i^J}$ is the number of sets W such that $|W| = i$ and the simulation of W along with J requires more than t shares of any of the input sharings. And $\mathbf{max}_J \, \mathbf{c_i^J}$ is the maximum over all possible collection of sets $J = (J_1, \ \ldots, \ J_m)$ of output wires indices such that $|J_1| \leq t, \ldots, |J_m| \leq t$.
This expression of ϵ is going to be used for our automatic verification tool of random probing security to also include the verification of (t, p, ϵ)-random probing composability for base gadgets. It will take as input a base gadget G and a parameter t such that $1 \leq t < n$, and compute bounds on the function $\epsilon = f(p)$ for which the circuit is (t, p, ϵ)-random probing composable. It also estimates upper and lower bounds on the tolerated leakage probability p by the gadget.

2.4 Random Probing Expandability

In addition to composing gadgets for global random probing security, we also exhibit a property that allows composable gadgets to be expanded, and achieve arbitrary security levels. This is called the expansion strategy and was originally introduced in [2]. The goal of the expansion strategy is to apply a standard circuit compiler several times on an arithmetic circuit, to reduce the simulation failure probability and achieve a desired security level. In [2], the expansion strategy relies on (m, c)-multiparty computation protocols, while our strategy relies on base n-share gadgets that fulfill what we call the random probing expandability property. Our advantage is that it is easier to achieve our property with base small gadgets, leading to efficient and explicit constructions.

The strategy consists in the following. Suppose that we have a standard circuit compiler CC with sharing order n and n-share gadgets G_{add}, G_{copy} and G_{mult}. We can apply the compiler CC on each of the base gadgets to obtain $G^{(2)}_{\text{add}} = CC(G_{\text{add}})$, $G^{(2)}_{\text{copy}} = CC(G_{\text{copy}})$, $G^{(2)}_{\text{mult}} = CC(G_{\text{mult}})$. This consists in replacing each basic gate in the compiled gadgets by the corresponding base gadgets G_{add}, G_{copy} and G_{mult} and each wire by n wires carrying an n-sharing of the original wire, giving us n^2-share gadgets for addition, multiplication and copy. This strategy can applied recursively a number of times to obtain n^k-share compiled gadgets $G^{(k)}_{\text{add}}$, $G^{(k)}_{\text{copy}}$, $G^{(k)}_{\text{mult}}$ to the level k. Expanding any arithmetic circuit C to the level k consists then in replacing each basic gate operation by the corresponding compiled gadgets $G^{(k)}_{\text{add}}$, $G^{(k)}_{\text{copy}}$, $G^{(k)}_{\text{mult}}$, and each wire by n^k wires carrying a n^k-sharing of the original wire. This strategy is associated to what we call an **Expanding Circuit Compiler**.

Definition 8 (Expanding Circuit Compiler). Let CC be the standard circuit compiler with sharing order n and base gadgets G_{add}, G_{mult}, G_{copy}. The **expanding circuit compiler** with **expansion level** k and base compiler CC is the circuit compiler $(CC^{(k)}, Enc^{(k)}, Dec^{(k)})$ satisfying:

- The input encoding $Enc^{(k)}$ is an (n^k)-linear encoding $LinEnc$.
- The output decoding Dec is the (n^k)-linear decoding mapping $LinDec$.
- The circuit compilation is defined as $CC^{(k)}(\cdot) = \underbrace{CC \circ CC \circ \cdots \circ CC}_{k \text{ times}}(\cdot)$

In fact, the expansion strategy allows to replace the leakage probability p of a wire in the original circuit by the failure event probability ϵ in the subsequent gadget simulation. If this simulation fails then one needs the full input sharing for the gadget simulation, which corresponds to leaking the corresponding wire value in the base case. This amplifies the security level, replacing the probability p in the base case by the probability ϵ as long as $\epsilon < p$. If the failure event probability ϵ can be expressed as a function of p: $\epsilon \leq f(p)$ for every leakage probability $p \in [0, p_{\max}]$ for some $p_{\max} < 1$, then the expanding circuit compiler with expansion level k shall result in a security amplification as

$$p = \epsilon_0 \xrightarrow{f} \epsilon_1 \xrightarrow{f} \cdots \xrightarrow{f} \epsilon_k = f^{(k)}(p) \ ,$$

which for an adequate function f (e.g. $f : p \mapsto p^2$) provides exponential security. In order to get such a security expansion, the gadgets must satisfy a stronger notion than the composability notion introduced in definition 7, which we call **random probing expandability**.
The definition of random probing expandability adds two additional conditions to the definition of random probing composability :

- Since in the base circuit before expansion, each wire as input of a gate leaks independently with probability p, a gadget should have a failure probability which is independent for each input.

- In case of failure event in the child gadget of an expanded circuit, the overall simulator should be able to produce a perfect simulation of the full output (that is the full input for which the failure occurs). To do so, the overall simulator is given the clear output (which is obtained from the simulation of the base circuit before expansion) plus any set of $n-1$ output shares. This means that whenever the set J is of cardinal greater than t, the gadget simulator can replace it by any set J' of cardinal $n-1$.

This gives us the following definition

Definition 9 (Random Probing Expandability). Let $f : \mathbb{R} \to \mathbb{R}$. An n-share gadget $G : \mathbb{K}^n \times \mathbb{K}^n \to \mathbb{K}^n$ is (t, f)-**random probing expandable (RPE)** if there exists a deterministic algorithm Sim_1^G and a probabilistic algorithm Sim_2^G such that for every input $(x', y') \in \mathbb{K}^n \times \mathbb{K}^n$, for every set $J \subseteq [n]$ and for every $p \in [0, 1]$, the random experiment

$$\begin{aligned} &W \leftarrow LeakingWires(G, p) \\ &(I_1, I_2, J') \leftarrow Sim_1^G(W, J) \\ &out \leftarrow Sim_2^G(W, J', x'|_{I_1}, y'|_{I_2}) \end{aligned}$$

ensures that

- the failure events $\mathcal{F}_1 \equiv \Big(|I_1| > t\Big)$ and $\mathcal{F}_2 \equiv \Big(|I_2| > t\Big)$ verify

$$Pr(\mathcal{F}_1) = Pr(\mathcal{F}_2) = \epsilon \quad \text{and} \quad Pr(\mathcal{F}_1 \wedge \mathcal{F}_2) = \epsilon^2 \tag{2.2}$$

  with $\epsilon = f(p)$ (in particular $\mathcal{F}_1$ and $\mathcal{F}_2$ are mutually independent),

- J' is such that $J' = J$ if $|J| \leq t$ and $J' \subseteq [n]$ with $|J'| = n-1$ otherwise,

- the output distribution satisfies

$$out \overset{\text{id}}{=} \Big(AssignWires(G, W, (x', y'))\ ,\ z'|_{J'}\Big) \tag{2.3}$$

where $z' = G(x', y')$.

For randomized arithmetic circuits, we have four types of gadgets : 1-to-1 (refresh gadgets), 1-to-2 (copy gadgets) and 2-to-1 (add and mult. gadgets). The definition of RPE for 2-to-1 gadgets

is given above in definition 9.
This definition can be simply extended to gadgets with 2 outputs: the Sim_1^G simulator takes two sets $J_1 \subseteq [n]$ and $J_2 \subseteq [n]$ as input and produces two sets J_1' and J_2' satisfying the same property as J' in the above definition (w.r.t. J_1 and J_2). The Sim_2^G simulator must then produce an output including $z_1'|_{J_1'}$ and $z_2'|_{J_1'}$ where z_1' and z_2' are the output sharings.
As for gadgets with 1 input, the Sim_1^G simulator produces a single set I so that the failure event ($|I| > t$) occurs with probability lower than ϵ (and the Sim_2^G simulator is then simply given $x'|_I$ where x' is the single input sharing).
This extends the RPE definition for 1-to-1 and 1-to-2 gadgets.

2.4.1 Expansion Security

It is not hard to see that the expandability property is stronger than the composability property since it adds two additional constraints to the definition of the latter.

Proposition 1. Let $f = \mathbb{R} \to \mathbb{R}$ and $n \in \mathbb{N}$. Let G be an n-share gadget. If G is (t, f)-RPE then G is (t, f')-RPC, with $f'(\cdot) = 2 \cdot f(\cdot)$.

When using expanded gadgets $G_{\text{add}}^{(k)}$, $G_{\text{copy}}^{(k)}$, $G_{\text{mult}}^{(k)}$ for the compilation of a base circuit C, we need to have the compiled circuit $\hat{C}$ to be random probing secure for certain parameters. The next theorem provides such a result, extending the expandability property from a base gadget G to its extended form $G^{(k)}$.

Theorem 2. Let $n \in \mathbb{N}$ and $f : \mathbb{R} \to \mathbb{R}$. Let G_{add}, G_{mult}, G_{copy} be n-share gadgets for the addition, multiplication and copy on $\mathbb{K}$. Let CC be the standard circuit compiler with sharing order n and base gadgets G_{add}, G_{mult}, G_{copy}. Let $CC^{(k)}$ be the expanding circuit compiler with base compiler CC. If the base gadgets G_{add}, G_{mult} and G_{copy} are RPE with $\epsilon = f(p)$ then, $G_{\text{add}}^{(k)} = CC^{(k-1)}(G_{\text{add}})$, $G_{\text{mult}}^{(k)} = CC^{(k-1)}(G_{\text{mult}})$, $G_{\text{copy}}^{(k)} = CC^{(k-1)}(G_{\text{copy}})$ are also RPE n^k-share gadgets for the addition, multiplication and copy on $\mathbb{K}$, with $\epsilon' = \epsilon^{(k)} = f^{(k)}$,

Using Proposition 1, and Theorems 1 and 2, we can deduce the random probing security of the expanding circuit compiler and any circuit compiled with it.

Corollary 1. Let $n \in \mathbb{N}$ and $f : \mathbb{R} \to \mathbb{R}$. Let G_{add}, G_{mult}, G_{copy} be n-share gadgets for the addition, multiplication and copy on $\mathbb{K}$. Let CC be the standard circuit compiler with sharing order n and base gadgets G_{add}, G_{mult}, G_{copy}. Let $CC^{(k)}$ be the expanding circuit compiler with base compiler CC. If the base gadgets G_{add}, G_{mult} and G_{copy} are (t, f)-RPE then $CC^{(k)}$ is $(p, 2 \cdot f^{(k)}(p))$-random probing secure. Equivalently, a compiled circuit $\hat{C}$ with the expanding circuit compiler is $(p, 2.|C|.f^{(k)}(p))$-random probing secure, where $|C|$ is the size of the original circuit.

Chapter 3

VRAPS : (V)erifier of (RA)ndom (P)robing (S)ecurity

In this chapter, I introduce our formal verification tool **VRAPS** which given a small gadget with wires leaking each with probability p, computes the parameters for which it is (p, ϵ)-random probing secure (RP), (t, p, ϵ)-random probing composable (RPC), and (t, f)-random probing expandable (RPE). Working on this tool from algorithms to implementations has taken a big part of my internship. I first describe the algorithm associated to the tool and how to extend it to verify the three properties RP, RPC and RPE. Next, I present my work on implementing the tool with efficient data representation, as well as optimizations that were added to be able to test relatively larger gadgets. My full code of the tool is made publicly available by CryptoExperts on their github page

https://github.com/CryptoExperts/VRAPS

3.1 VRAPS Formal Algorithm

3.1.1 Random Probing Security Verification Algorithm

As stated in Section 2.2.1, when a compiled circuit $\hat{C}$ with s wires is (p, ϵ)-RP secure, then ϵ can be computed as a function $f(p)$ as

$$\epsilon = f(p) = \sum_{i=1}^{s} c_i \; p^i (1-p)^{s-i}$$

So computing $f(p)$ amounts to computing the coefficients c_i of $f(p)$ which are the number of sets W of size i of leaking wire labels for which there is a simulation failure. For each c_i, it is clear

that there are $\binom{s}{i}$ possible sets of wires to test. Computing all coefficients $c_1, \ldots, c_s$ is exponential (lower bounded by 2^s). For that, our algorithm consists in computing the coefficients $c_1, \ldots, c_\beta$, where $1 < \beta \leq s$ is a threshold on the number of coefficients which controls the verification duration. This procedure gives Algorithm 1, which computes a lower bound on $f(p)$. An upper bound on $f(p)$ is obtained be replacing each coefficient c_i for $i > \beta$, by the binomial coefficient $\binom{s}{i}$.

Function listTuples outputs the list of all sets of wire labels of size i from the total s wires

Algorithm 1 VRAPS: RP Security Verification

Input: a compiled circuit $\hat{C}$ with s wires and a threshold $\beta \leq s$
Output: a list of β coefficients c_1, ..., c_β

1: $(c_1, \ldots, c_\beta) \leftarrow (0, \ldots, 0)$
2: **for** $i = 1$ to β **do**
3: $\quad L_{W_i} \leftarrow \mathsf{listTuples}(s,i)$ ▷ list of $\binom{s}{i}$ possible sets W of i wire labels
4: $\quad L_p \leftarrow \mathsf{failureTest}(\hat{C}, L_{W_i})$ ▷ identify failure tuples in L_{W_i}, and store them in L_p
5: $\quad c_i \leftarrow \mathsf{SizeOf}(L_p)$
6: **end for**
7: **return** $(c_1, \ldots, c_\beta)$

and stores them in the list L_{W_i}. Then function failureTest takes as input the latter list and the compiled circuit $\hat{C}$, and checks for each set W of wire labels in L_{W_i}, whether it can be perfectly simulated without the knowledge of n shares of any of the inputs. Basically, for each set of wires W, a sequence of rules inspired from **maskVerif** [4] is applied to determine whether W can be perfectly simulated (**maskVerif** is a tool for automatic verification of security of circuits in the probing model). For that, each wire w_i labeled in W is considered together with the algebraic expression $\varphi_i(\cdot)$ describing its assignment by $\hat{C}$ as a function of the circuit inputs and the random values returned by the random gates, then the three following rules are successively and repeatedly applied on all the wire sets W in L_{W_i}:

rule 1: check whether all the expressions $\varphi_i(\cdot)$ corresponding to wires w_i in W contain all the shares of at least one of the coordinates of the input sharings x';
Example (2-share gadget of inputs x, y): Consider $W = \{(x_0 + y_0), (x_1)\}$ with 2 wires. It is clear that W contains both shares of the input x, so it cannot be perfectly simulated without the knowledge of the full input x. So W represents a potential failure.

rule 2: for every $\varphi_i(\cdot)$, check whether a random r (i.e. an output of a random gate) additively masks a sub-expression e (which does not involve r) and appears nowhere else in the other $\varphi_j(\cdot)$ with $j \neq i$; in this case replace the sum of the so-called sub-expression and r by r,

namely $e + r \leftarrow r$;

Example (2-share gadget of inputs x, y): Consider $W = \{(x_0 + r_0 + r_1), (x_1 + r_1), (y_0)\}$ with 3 wires, which contains both shares of the input x, so W is a potential failure set. It is clear that the random value r_1 appears twice in the first and second wire expressions. Meanwhile, the random value r_0 appears only once in the first wire expression. So r_0 can be used to mask the first wire expression. W is thus equivalently replaced by $W' = \{(r_0), (x_1 + r_1), (y_0)\}$. Now in W', r_1 also appears only once, so W' can be replaced by $W'' = \{(r_0), (r_1), (y_0)\}$. Which means that W is eventually not a failure set, by application of rule 2.

rule 3: apply mathematical simplifications on the tuple.

Example (2-share gadget of inputs x, y): Consider $W = \{(x_0+y_0+r_0), (x_0+y_0+r_0+x_1+r_1)\}$ with 2 wires. Then W can be simplified and replaced by an equivalent set $W' = \{(x_0 + y_0 + r_0), (x_1 + r_1)\}$, replacing the second wire from W by the sum of both its wires in W'. This reduces the total number of variables manipulated in the considered set W', making it easier to apply the other rules and test for failure sets.

3.1.2 Random Probing Composability Verification Algorithm

As explained in Section 2.3.1, for parameters $p \in [0, 1]$ and $1 \leq t < n$, computing the failure simulation probability ϵ amounts to computing $f(p)$ such that

$$f(p) = \sum_{i=1}^{s} \ \mathbf{max}_J \, \mathbf{c_i^J} \ \ p^i (1-p)^{s-i}$$

where J is a collection of sets $J = (J_1, \ \ldots, \ J_m)$ of t output wire indices for each output sharing, and where $\mathbf{c_i^J}$ is the number of sets W such that $|W| = i$ and the simulation of W along with J requires more than t shares of any of the input sharings. $\mathbf{max}_J \, \mathbf{c_i^J}$ is the maximum over all possible collection of sets J. Then Algorithm 1 can be simply extended to verify RPC property by adding the output wires to the simulation, as described in Algorithm 2.

Function listOutputTuples enumerates all possible sets of t output wire indices $J_1, \ldots, J_m$ for all output sharings, and then outputs all possible collection of sets $J = (J_1, \ldots, J_m)$. Then function failureTest2 applies the same set of rules as failureTest, but considers each set of wire labels W from L_{W_i}, concatenated with the list of output wire labels from the set J. In this case, it will consider a simulation failure if W and J require the knowledge of at least $t + 1$ shares of any of the input sharings, contrarily to failureTest which considers a failure when n shares are needed of any of the input sharings. The resulting coefficients c_i are then taken as the maximum of all coefficients computed by considering each J in L_J.

Algorithm 2 VRAPS: RPC Security Verification

Input: a l-to-m n-share gadget G with s wires, a threshold $\beta \leq s$ and a parameter $1 \leq t < n$.
Output: a list of β coefficients c_1, ..., c_β

$(c_1, \ldots, c_\beta) \leftarrow (0, \ldots, 0)$
$L_J \leftarrow \mathsf{listOutputTuples}(n,t,m)$ ▷ list all possible collection of sets J of t output wire indices for each of the m output sharings
for J in L_J **do**
  $(c'_1, \ldots, c'_\beta) \leftarrow (0, \ldots, 0)$
  for $i = 1$ to β **do**
    $L_{W_i} \leftarrow \mathsf{listTuples}(s,i)$ ▷ list of $\binom{s}{i}$ possible sets W of i wire labels
    $L_p \leftarrow \mathsf{failureTest2}(G, L_{W_i}, J)$ ▷ identify failure tuples in L_{W_i}, and store them in L_p
    $c'_i \leftarrow \mathsf{SizeOf}(L_p)$
  end for
  $(c_1, \ldots, c_\beta) \leftarrow \left(\max(c_1, c'_1), \ \ldots, \ \max(c_\beta, c'_\beta) \right)$
end for
return $(c_1, \ldots, c_\beta)$

3.1.3 Random Probing Expandability Verification Algorithm

Verifying the (t, f)-RPE property is trickier to include in the tool. We will consider in the following the case with 2-to-1 gadgets, which can be easily extended to 1-to-2 and 1-to-1 gadgets.
First, the requirement of definition 9 that the failure events $\mathcal{F}_1$ and $\mathcal{F}_2$ are mutually independent might seem too strong. Which is why, in practice, it might be easier to show or verify that some gadgets satisfy a weaker notion that we call **weak RPE** (wRPE). The definition of (t, f)-wRPE is the same as the definition of RPE in definition 9, except for the probabilities of the failure events. Instead, we say that a gadget is (t, f)-wRPE if the failure events verify $Pr(\mathcal{F}_1) \leq \epsilon$, $Pr(\mathcal{F}_2) \leq \epsilon$ and $Pr(\mathcal{F}_1 \wedge \mathcal{F}_2) \leq \epsilon^2$. The wRPE property is thus easier to verify by our tool than RPE. In addition, there is a tight reduction from wRPE to RPE.

Proposition 2. Let $f = \mathbb{R} \to \mathbb{R}$. Let $G : \mathbb{K}^n \times \mathbb{K}^n \to \mathbb{K}^n$ be an n-share gadget. If G is (t, f)-wRPE then G is (t, f')-RPE with $f'(\cdot) = f(\cdot) + \frac{3}{2} f(\cdot)^2$.

The reduction easily extends to 1-to-2 and 1-to-1 gadgets.
Then if we can verify using our tool that a gadget is wRPE with

$$Pr(\mathcal{F}_1) = f_1(p) \ , \quad Pr(\mathcal{F}_2) = f_2(p) \ \text{ and } \ Pr(\mathcal{F}_1 \wedge \mathcal{F}_2) = f_{12}(p) \ ,$$

Then the gadget would be (t, f)-RPE with

$$f : p \mapsto f_{\max}(p) + \frac{3}{2} f_{\max}(p)^2 \ \text{ with } \ f_{\max} = \max(f_1, f_2, \sqrt{f_{12}}) \ .$$

Thus, computing the function $f(p)$ for (t, f)-RPE, amounts to computing failure functions on the first input f_1, the second input f_2 and on both at the same time f_{12}, and then taking the maximum between the first two functions and the square root of the third.
The verification algorithm is split into three steps with respect to the size of the set J of indices of output shares (as in definition 9):

1. $|J| \leq t$. During this step, the verification algorithm considers all possible sets J' of t output share indices, giving the corresponding functions $f_1^1,\ f_2^1, f_{12}^1$. It is almost the same algorithm as Algorithm 2 for RPC verification, with the difference of computing three failure functions at the same time, and instead of considering failure when $t+1$ shares or more of **any** input sharing are needed, splits the failure sets into failures on the first input ($t+1$ or more shares of the first input are needed), and on the second one ($t+1$ or more shares of the second input are needed). Failure sets on both inputs are just the intersection of both failure sets on the first and on the second. The complete algorithm for this step (RPE1) is given in appendix A, Algorithm 3. We shall call this step in the rest of the report as (t, f^1)-RPE1 property, with $f^1 = \max(f_1^1, f_2^1, \sqrt{f_{12}^1})$.

2. $|J| > t$. During this step, the verification algorithm searches for at least one set of output shares J' of $n-1$ shares for which there is a simulation success for each set of wires W. If none is found, then W is considered as a failure set. This gives the corresponding functions $f_1^2,\ f_2^2, f_{12}^2$. In this algorithm, the coefficients c_i are updated only with sets of wires W for which there is a simulation failure with all sets of $n-1$ output share labels. This means that for this set W, there is no set of $n-1$ output shares J for which the simulation succeeds, which does not fulfill the condition from definition 9, resulting indeed in a failure set for RPE2. The failures are also split into failures on each input and on both inputs. The complete algorithm for this step (RPE2) is given in appendix A, Algorithm 4. We shall call this step in the rest of the report as (t, f^2)-RPE2 property, with $f^2 = \max(f_1^2, f_2^2, \sqrt{f_{12}^2})$.

3. The final function f for RPE will be

$$f = \max\left(f_1^1, f_1^2, f_2^1, f_2^2, \sqrt{f_{12}^1}, \sqrt{f_{12}^2}\right)$$

   Or $f = \max(f^1, f^2)$.
   The final procedure for RPE is given in appendix A, Algorithm 5, which calls the verification functions of Algorithms 3 and 4.

Remark 2. The computation of f for RPE for 2 outputs gadgets is a bit trickier. Instead of two verification steps considering both possible ranges of cardinals for the output set of shares J, we

need to consider four scenarios for the two possible features for output sets of shares J_1 and J_2. In a nutshell, the idea is to follow the first verification step described above when both J_1 and J_2 have cardinal equal or less than t and to follow the second verification step described above when both J_1 and J_2 have greater cardinals. This leads to functions $f^{(1)}$ and $f^{(2)}$. Then, two extra cases are to be considered, namely when ($|J_1| \leq t$) and ($|J_2| > t$) and the reverse when ($|J_1| > t$) and ($|J_2| \leq t$). To handle these scenarios, our tool loops over the output sets of shares of cardinal equal or less than t for the first output, and it determines whether there exists a set of $n-1$ shares of the second output that a simulator can perfectly simulate with the leaking wires and the former set. This leads to function $f^{(12)}$ and reversely to function $f^{(21)}$. From these four verification steps, we can deduce f such that the copy gadget is (t, f)-wRPE:

$$\forall p \in [0,1], f(p) = \max(f^{(1)}(p), f^{(2)}(p), f^{(12)}(p), f^{(21)}(p)).$$

3.2 Data Representation & Optimizations

When moving to the implementation of the verification algorithm for our tool VRAPS, it was clear that we needed to add some optimizations and find a good representation of the wires and gates data. The computation of the coefficients c_i in all of the above algorithms becomes rapidly exponential as the number of wires grow. And since we fix a threshold β on the computed coefficients, the bounds on the function $f(p)$ that we obtain may not be very *tight* when β is small. Indeed, this tool is interesting for testing security of small gadgets, which is proven interesting in the state-of-the-art, for instance to verify gadgets and then deduce global security through composition properties and/or low-order masked implementations. However, through the following data representation techniques and optimizations, we try to increase the bounds on this threshold β that we can fix while maintaining a reasonable verification time.

The whole tool is implemented in sageMath on python 3. A first basic implementation of the tool was provided by my supervisors, with an aim to have a "working" tool. In the following, I will present the version I implemented of the tool, adding several theoretical and implementation optimizations, and changing the data representation to be able to use advanced python libraries such as numpy for arrays manipulation. I will also provide an efficiency comparison between the basic implementation and my new implementation. The latter is made publicly available on github.

3.2.1 Optimizations

Several optimizations have been considered in order to reduce the verification time of the algorithms for RP, RPC and RPE security.

Grouping the wires. In a circuit C, several wires can have the same algebraic expression. And since when applying the rules for testing for failure sets of wires, we consider these algebraic expressions, it is interesting to be able to "group" these wires in a same variable with a certain number of occurrences. For example, if a variable x is used k times in a circuit, then this means that it is automatically copied into k variables. Since each copy gate takes 1 input wire and outputs 2 copies of the input, then this means that we will have in the circuit $k-1$ copy gates and $2k-1$ wires that have the exact same algebraic expression x. Grouping this kind of wires allows to reduce the number of wires to consider in the circuit (a circuit C with s wires that has a variable used k times, will be reduced in the verification algorithm to a circuit C' with $s-2k+1$ wires with a wire labeled with $2k-1$ occurrences). This way, we consider each wire w_i by its expression, and we associate it with a metric n_i which is the number of wires that have this expression. Meanwhile, this means that updating the coefficients in each iteration should also be altered. In fact, consider a set $W = \{(w_1, 1), (w_2, 2), (w_3, 3)\}$ of 3 wires that was found to be a failure set (using any of the function failureTest, failureTest2, failureTest3), where each wire is associated with its number of occurrences in the circuit. In the base case, this would result in adding 1 to the coefficient c_3 ($c_3 = c_3 + 1$). However, in this case, we need to update c_3 with 6, c_4 with 9, c_5 with 3 and c_6 with 1, since we can consider any combination of the wires and their occurrences. The latter evaluation is performed using a recursive function which evaluates the number of partitions of an integer j into h parts with the constraints that each part should be at least one. It can be seen that this optimization also allows to update coefficients c_j for $j > \beta$, providing a tighter lower bound on the computed function f whether it is for RP, RPC or RPE.

Computation in batches. For each iteration $i = \{1, \ldots, \beta\}$ in the verification algorithm, we have a total of $\binom{s}{i}$ sets of wires W to consider. Listing all these sets at once with the function listTuples, causes memory issues when the values of i and s are sufficiently large. To avoid having constraints on RAM size, we decide to list sets of wires in batches. In other terms, instead of listing all $\binom{s}{i}$ sets at once to look for failure sets, we list the sets progressively in batches of fixed size m. This results in splitting each iteration i of the procedure into $\left\lceil \binom{s}{i}/m \right\rceil$ iterations for all batches. Indeed, with this optimization, the only constraint that we still have is on the verification time, since we can split very large lists using batches and not cause any memory issues.

Using incompressible sets. This optimization is better explained with an example. Consider Algorithm 1. Let $W = \{w_1, w_2\}$ be a set of 2 wires of a circuit C with s wires, which function failureTest found to be a failure set on iteration $i = 2$. Then on iteration $i = 3$, the set $W' = \{w_1, w_2, w_j\}$ is automatically a failure set for any $j \in \{3, \ldots, s\}$. The same goes on iteration $i = 4$ for $W' = \{w_1, w_2, w_j, w_h\}$ for any $j, h \in \{3, \ldots, s\}$, ... If the subsets $\{w_1\}$ and $\{w_2\}$ are not failure sets, then W is said to be an **incompressible failure set**, and can be used to eliminate sets W' of size bigger than 2 and that contain the set W, before passing them to the function failureTest, to avoid verification time loss on applying the simplification rules. This optimization is thus integrated in Algorithm 1 as follows :

- At the beginning, a list of incompressible failure sets is initiated $L_{inc} \leftarrow []$.
- On each iteration $i = \{1, \ldots, \beta\}$, sets from L_{W_i} that contain any subset from L_{inc} are removed from L_{W_i}, and the corresponding coefficients c_i are updated according to the removed sets immediatly.
- The remaining elements from L_{W_i} are passed to failureTest, and then elements from L_p are added to L_{inc}, $L_{inc} \leftarrow L_{inc} + L_p$. The coefficients c_i are also updated according to elements from L_p.
- Move to next iteration $i + 1$.

Remark 3. The above optimization of incompressible sets is also used for RPC verification in Algorithm 2. However, it cannot be directly used for RPE verification, since failure sets are considered on each input independently, which also has to be taken into account for incompressible failure sets. For this reason, during my internship, I implemented this optimization for the algorithms of random probing security and RPC verification, but not for RPE verification.

3.2.2 Data Representation

Input circuit files are sage files that are composed of a list of instructions with + and × operations. We consider by default that we are in a boolean polynomial ring. Figure 3.1 shows an example of an input circuit file on the well-known ISW multiplication gadget from [17]. The first lines of the input file specifies the number of shares n, names of the l input variables, all random variables, and names of the m output variables, for an n-share l-to-m gadget. Share numbers range from 0 to $n-1$. This is followed by the list of instructions representing intermediate and output variables or what we call the wires. Necessary copy gates are implicitly added while reading the file, when

```
#SHARES 2
#IN a b
#RANDOMS r0
#OUT d

c0 = a0 * b0
d0 = c0 + r0

c1 = a1 * b1
c1 = c1 + r0
tmp = a0 * b1
c1 = c1 + tmp
tmp = a1 * b0

d1 = c1 + tmp
```

Figure 3.1: Input Circuit file for 2-share ISW multiplication gadget [17].

a wire is used more than once.
To execute the verification algorithm, each wire w is represented as a tuple of values as follows

$$w = (\mathsf{algebraic\_expression}, \mathsf{secret\_deps}, \mathsf{random\_deps}, \mathsf{nb\_occs}, \mathsf{weight})$$

- algebraic_expression: This is a string of the developed algebraic expression in the boolean polynomial ring of the wire w as described earlier.

- secret_deps: This is an array of l integers each corresponding to an input variable, all initialized to 0. For each input variable x, for each sharing number $i \in \{0, \ldots, n-1\}$ of x that is in the expression of the wire w, we add the value 2^i to the corresponding integer in secret_deps. This representation allows us to rapidly add the secret dependencies of several wires (binary OR) and test for failures (a failure for RP for example on a variable x is if the corresponding integer is equal to $2^n - 1$).
  Example: Consider a 3-share circuit C and a wire w of C. If C is a circuit with 2 inputs a and b, then secret_deps $= [0, 0]$ (first element corresponds to a, and the second corresponds to b). If $w = (a_0 + a_1 \times b_2)$, then w will have the list secret_deps $= [2^0 + 2^1,\ 2^2] = [3,\ 4]$.

- random_deps: This is an array of integers each corresponding to a random variable. The corresponding value is equal to 0 if the random value does not appear in the expression of w, otherwise it is equal to 1. This representation allows us to rapidly add the random

dependencies of several wires (small integer addition).
Example: Consider a circuit C and a wire w of C. If C is a circuit with a total of 3 random variables r_0, r_1, r_2, then random_deps $= [0, 0, 0]$. If $w = (r_0 + r_2)$, then w will have the list random_deps $= [1,\ 0,\ 1]$.

- nb_occs: This field corresponds to the number of wires in the original circuit that have the same algebraic expression, to be used for the optimization of **grouping the wires** as explained earlier.

- weight: this field is to be used for the optimization of **incompressible sets**. In fact, each wire w has a unique index ind in the list of wires. Its weight is the value 2^{ind}. A set of wires W will have a weight equal to the sum of its wires weights. Then, we can easily test if a set of wires W' of weight W'_h contains an incompressible failure set of wires W of weight W_h, by using a binary AND operation (if $W'_h \ \& \ W_h = W_h$ then $W \subseteq W'$).

The fields secret_deps and random_deps allow to efficiently implement the simplification rules of the functions failureTest, failureTest2, failureTest3, since all rules amount now to performing binary operations on arrays of integers. In addition, the fields nb_occs and weight allow to efficiently implement the optimizations from Section 3.2.1 using binary tests and operations.

Remark 4. In the first implementation of the algorithm provided by my supervisors, each wire was represented by its name, algebraic expression and its weight since they also implemented the technique using **Incompressible sets**. In my implementation, I slightly modified these fields of the tuple to use it in my code, and added the other fields mentioned earlier (secret_deps, random_deps, nb_occs).

To efficiently manipulate all of the values, I used in my implementation the python library numpy (the first implementation used traditional python arrays and loops manipulation). This is a library that is usually used for machine learning purposes since it offers fast data arrays manipulation. For that, each circuit is associated with 6 numpy arrays , where the first one is the array of wire indices from 0 to $s - 1$, and the other ones are the corresponding values from the wires reprensetation (the arrays for secret_deps and random_deps are in fact numpy matrices). This allows us to efficiently perform reduction operations between several values, including integer and binary operations necessary for application of the simplification rules, and the optimizations.

Table 3.1: Verification time (in s, on an Intel i7-8550U CPU, 8GB RAM) of **VRAPS** for (p, ϵ)-RP property, for different n-share circuits and different number of wires.

n	# wires	# variables	β	**RP** Verification Time (in seconds)	
				First Implem.	Optimized Implem.
2	19	13	13	1.6	0.23
3	36	24	7	807.36	19.69
4	48	32	7	1000.12	50.84
5	122	82	6	**Memory Error on** c_6	46.29

3.3 Experimental Results

The tool has been tested on different n-share circuits and gadgets and with different number of wires. Table 3.1 shows the execution time for Algorithm 1 for (p, ϵ)-RP verification, using its first implementation by my supervisors, which is the first version of the tool, and my implementation taking into account the optimizations and data manipulation from Section 3.2. Different circuits have been tested with a sharing order of up to 5. The column *#wires* represents the total number of wires in the circuit, while *#variables* is the number of wires with distinct algebraic expressions, taking into account the optimization of **Grouping the wires** from section 3.2.1. First, observe that as the size of the circuit grows, the verification time increases exponentially, which directly results from the exponential nature of the algorithm (Recall that this tool is interesting for testing small circuits). The value of the threshold β is fixed in a way to have a desired bound on the execution time, and as the value of β grows, the verification time will also exponentially increase. Meanwhile, the speedup of my implementation over the first version of the tool is clear from the execution times in the table. Moreover, in the case of a 5-share circuit with 122 wires, enumerating all of the sets of wires of size 6 for the coefficient c_6 causes a memory overflow (on a **8GB RAM** computer) for a total of $\binom{122}{6}$ wires, which confirms the interest of using the **batching** technique introduced earlier, in order to overcome this issue. With this technique, the algorithm is only limited by its verification time. It was also observed during the execution of the tool, that the main computational cost of the algorithm is in the application of the simplification rules for the search of failure sets of wires.

The speedup of my implementation is also observable for RPC and RPE verifications. In fact, in the verification algorithms of these properties, the same set of rules as for RP verification is applied on all sets of wires from size 1 to size β. The difference is that this application is repeated for all possible sets of output shares indices J. From this, it is clear that these verifications will be more costly than RP verification. Table 3.2 shows the execution time for Algorithm 5 for

Table 3.2: Verification time (in s, on an Intel i7-8550U CPU, 8GB RAM) of **VRAPS** for (t, f)-RPE property, for different n-share circuits and different number of wires.

n	# wires	# variables	β	t	**RPE** Verification Time (in seconds)	
					First Implem.	Optimized Implem.
2	19	13	13	1	223.11	18.24
3	36	24	4	1	96.46	15.48
				2	101.2	5.78
4	48	32	4	1	800.94	47.22
				2	147.33	8.65
				3	82.42	6.22

(t, f)-RPE verification (we don't include results for Algorithm 2 for RPC verification since the first step of RPE verification (RPE1) in Algorithm 3 is almost the same as Algorithm 2, so the execution time already gives an idea about both verification costs). We can see from Table 3.2 that the speedup of my implementation over the first implemetation is more interesting than for RP verification in Table 3.1, this is because as stated above, the RPE verification algorithm repeats the application of the set of rules for all possible sets of output shares. Observe also that the value of β fixed to have the same execution times as in Table 3.1 is smaller, since there is more computation involved in RPE verification. When the value of β grows, the execution time will grow exponentially, and my implementation allows to fix larger values of β while maintaining a reasonable execution time. Unlike the first implementation, where for example for 4 shares, the value of $\beta = 4$ and $t = 1$ already holds a verification time of 800 seconds. When $\beta > 4$, this execution will be very costly.

Note also that we have the same behavior for RPE verification as for RP verification, where the larger the number of shares and the number of wires, the more computationally difficult it becomes to compute the coefficients c_i, which is very expected.

Chapter 4

Comparison with AIS [2], Instantiation, Compiled AES

Our definition of the expansion strategy and RPE property from Section 2.4 is inspired from the work from [2], which provides a definition of a circuit compiler (the AIS compiler) that aims to amplify security levels like in our case. In this chapter, I will provide a complexity analysis of our expansion strategy which was established by my supervisors. Then, another mission of my internship consisted in providing an analogous analysis of the AIS compiler to compare it with the one from our work. I will also describe this analysis and compare it with the complexity of our expansion strategy, providing an instantiation for both strategies to have a concrete complexity comparison. Then, I will exhibit my implementation of the AES encryption algorithm using our instantiation, as an example of usage of the expansion strategy in a real-life scenario. This implementation is publicly available on CryptoExperts' github page

https://github.com/CryptoExperts/poc-expanding-compiler

4.1 Complexity Analysis

4.1.1 Expanding Circuit Compiler Complexity

In this section, I will show that compiling a circuit C with the expanding circuit compiler $CC^{(k)}$ from definition 8, of sharing order n, and base gadgets G_{add}, G_{copy}, G_{mult}, gives a circuit $\hat{C} = CC^{(k)}(C)$ of complexity $|\hat{C}| = \mathcal{O}(|C|.\kappa^e)$, where $|C|$ is the size of the original circuit, κ is a security parameter and e is a certain exponent.
To do this, we represent each circuit or gadget G by a vector $N = (N_a, N_c, N_m, N_r)$ with $|G| = N_a + N_c + N_m + N_r$, where N_a is the number of **addition gates** in the gadget, N_c the number of **copy gates**, N_m the number of **multiplication gates**, and N_r the number of **random gates**.

Let N_{add}, N_{copy}, N_{mult} and N_{rand} be the gates count vectors corresponding to each of the base gagdets. As stated in Remark 1, $N_{rand} = (0, 0, 0, n)$. Let us construct the following matrix M

$$M = \begin{array}{c} \begin{matrix} N_{add}^T & N_{copy}^T & N_{mult}^T & N_{rand}^T \end{matrix} \\ \begin{pmatrix} N_{add,a} & N_{copy,a} & N_{mult,a} & 0 \\ N_{add,c} & N_{copy,c} & N_{mult,c} & 0 \\ N_{add,m} & N_{copy,m} & N_{mult,m} & 0 \\ N_{add,r} & N_{copy,r} & N_{mult,r} & n \end{pmatrix} \end{array} \tag{4.1}$$

Let C be a circuit of gates count vector N_C to be compiled with $CC^{(k)}$. It can be checked that the compiled circuit $\hat{C} = CC^{(k)}(C)$ for an expansion level k holds a vector $N_{\hat{C}} = M^k.N_C$, with $M^k = (\ {N_{add}^T}^k \mid {N_{copy}^T}^k \mid {N_{mult}^T}^k \mid {N_{rand}^T}^k\ )$ for k expanded base gadgets $G_{\text{add}}^{(k)}$, $G_{\text{copy}}^{(k)}$ and $G_{\text{mult}}^{(k)}$.
Using the eigen decomposition of the matrix $M = Q.\Lambda.Q^{-1}$, we get

$$M^k = Q.\Lambda^k.Q^{-1} \quad , \quad with \ \ \Lambda^k = \begin{pmatrix} \lambda_1^k & & & \\ & \lambda_2^k & & \\ & & \lambda_3^k & \\ & & & \lambda_4^k \end{pmatrix} \tag{4.2}$$

So the complexity of a compiled circuit $\hat{C}$ with expansion level k, can be expressed as

$$|\hat{C}| = \mathcal{O}\Big(|C|.\max(\lambda_1, \lambda_2, \lambda_3, \lambda_4)^k\Big) = \mathcal{O}\Big(|C|.N_{\max}^k\Big) \tag{4.3}$$

As exhibited in Section 2.4, the goal of the expansion strategy is to replace the leakage probability p of a wire in the circuit C, by the failure event probability $\epsilon = f(p)$ in the compiled circuit $\hat{C}$. To attain a certain security level κ, we apply the expansion strategy k times to amplify the security level and achieve a failure event probability ϵ^k such that $\epsilon^k = f^k(p) \leq 2^{-\kappa}$. Such a function f is actually of the form

$$f(p) = \sum_{i \geq d} c_i p^i \leq \Big(c_d + \mathcal{O}(p)\Big)p^d = c'_d p^d$$

Informally, d is the smallest size of sets of wire labels W, for which there is a simulation failure for the RPE property (check Algorithm 5). We shall call d the **amplification order** of f.
So to achieve a security level κ, we need to have $f^k(p) < (c'_d p)^{d^k} \leq 2^{-\kappa}$. Assuming that $c'_d p < 1$, this gives us $k \geq \log_d(\kappa) - \log_d(-\log_2(c'_d p))$. Then the complexity from equation (4.3) can be

rewritten as

$$|\hat{C}| = \mathcal{O}\Big(|C|.\kappa^e\Big) \quad , \quad with \;\; e = \frac{\log(N_{max})}{\log(d)} \tag{4.4}$$

4.1.2 AIS Compiler [2] Complexity

The work from [2] provides a construction of a circuit compiler (the AIS compiler) based on the expansion strategy with a (p, ϵ)-composable security property, analogous to our (t, f)-RPE property. To this purpose, the authors use an (m, c)-multi-party computation (MPC) protocol Π. Such a protocol allows to securely compute a functionality shared among m parties and tolerating at most c corruptions. In a nutshell, their composable circuit compiler consists of multiple layers: the bottom layer replaces each gate in the circuit by a circuit computing the (m, c)-MPC protocol for the corresponding functionality (either Boolean addition, Boolean multiplication, or copy). The next $k-1$ above layers apply the same strategy recursively to each of the resulting gates. As this application can eventually have exponential complexity if applied to a whole circuit C directly, the top layer of the compilation actually applies the k bottom layers to each of the gates of C independently and then stitches the inputs and outputs using the correctness of the XOR-encoding property. Hence the complexity is in

$$\mathcal{O}(|C|.N_g^k) \;, \tag{4.5}$$

where $|C|$ is the number of gates in the original circuit to be compiled and N_g is the number of gates in the circuit computing Π. The authors of [2] prove that such compiler satisfies (p, ϵ)-composition security property, where p is the tolerated leakage probability and ϵ is the simulation failure probability. Precisely:

$$\epsilon = N_g^{c+1}.p^{c+1} \tag{4.6}$$

Equations (4.5) and (4.6) can be directly plugged into our asymptotic analysis of Section 4.1.1, with N_g replacing our $N_{\max}$ and where $c+1$ stands for our amplification order d. The obtained asymptotic complexity for the AIS compiler is

$$\mathcal{O}\Big(|C|.\kappa^e\Big) \quad , \quad with \;\; e = \frac{\log(N_g)}{\log(c+1)} \tag{4.7}$$

This is to be compared to $e = \dfrac{\log(N_{\max})}{\log(d)}$ in our scheme. Moreover, this compiler can tolerate a leakage probability

$$p = \frac{1}{N_{\mathrm{g}}^2} \;.$$

4.2 Instantiations

4.2.1 RPE Instantiation

For our expansion strategy, we provide a construction of 3-share base gadgets G_{add}, G_{copy} and G_{mult}, and we use them to instantiate our expanding circuit compiler. To benefit from the expansion strategy, we need a failure event probability function $f(p)$ of **amplification order** strictly greater than 1 as explained in Section 4.1.1, to guarantee that there exists a probability $p_{max} \in [0, 1]$ such that $\forall p \leq p_{max},\ f(p) \leq p$.

We first observe that concerning 2-share gadgets, there are no (2-share, 2-to-1) $(1, f)$-RPE gadgets such that f has an amplification order greater than one. We give a detailed proof of this statement in appendix B.

Because of this result, we provide constructions with 3 shares instead. We are able to construct 3-share gadgets G_{add}, G_{copy} and G_{mult} that are $(t = 1, f)$-wRPE. A complete description of these gadgets is given in appendix B.

In a nutshell, we use our tool **VRAPS** to show that these gadgets are $(t = 1, f)$-wRPE. Our tool outputs a maximum failure function for all of the gadgets

$$f(p) = \sqrt{83}p^{3/2} + \mathcal{O}(p^2)$$

of amplification order $\dfrac{3}{2}$. The complexity analysis of our gadgets holds a matrix

$$M = \begin{array}{c} \begin{matrix} N^T_{add} & N^T_{copy} & N^T_{mult} & N^T_{rand} \end{matrix} \\ \begin{pmatrix} 15 & 12 & 28 & 0 \\ 6 & 9 & 23 & 0 \\ 0 & 0 & 9 & 0 \\ 6 & 6 & 11 & 3 \end{pmatrix} \end{array}$$

and shows that the value of N_{max} is equal to 21, giving us a total complexity

$$|\hat{C}| \ = \ \mathcal{O}(|C|.\kappa^{7.5}) \ \ , \ \ with \ \ e = 7.5 = \frac{\log(21)}{\log(3/2)}$$

The maximum tolerated leakage probability defined as the maximum value p such that $f(p) \leq p$ is of $p_{\max} \approx 2^{-8}$.

4.2.2 AIS Compiler Instantiation

As for the AIS compiler, the authors from [2] provide an instantiation with an existing multiparty computation (m, c)-MPC protocol due to Maurer [19]. By analyzing the instantiation, we observe that this protocol can be implemented with a circuit of N_{g} gates with

$$N_{\mathrm{g}} = (6m-5) \cdot \left(\binom{m-1}{c}^2 + m(2k-2) + 2k^2 \right) \quad \text{where} \quad k = \binom{m}{c} .$$

We provide the full analysis in appendix C.
They instantiate their compiler with this protocol for parameters $m = 5$ parties and $c = 2$ corruptions, from which they get $N_{\mathrm{g}} = 8150$. This yields a tolerated leakage probability of $p = \frac{1}{8150^2} \approx 2^{-26}$ and an exponent $e = \log 8150 / \log 3 \approx 8.19$ in the asymptotic complexity $\mathcal{O}\left(|C|.\kappa^e\right)$ of the AIS compiler from section 4.1.2.

These results are to be compared to the $p \approx 2^{-8}$ and $e \approx 7.5$ achieved by our construction. We note that our construction achieves a slightly better complexity while tolerating a much higher leakage probability.

Remark 5. Further instantiations of the AIS scheme (based on different MPC protocols) or of our scheme (based on different gadgets) could lead to better asymptotic complexities and/or tolerated leakage probabilities.

4.3 Instantiation with AES Implementation

We provide an implementation in sageMath in python of the expanding compiler CC, which given a sharing order n, base gadgets G_{add}, G_{copy}, G_{mult} and an expansion level k, outputs the expanded gadgets $G^{(k)}_{\mathrm{add}}$, $G^{(k)}_{\mathrm{copy}}$,$G^{(k)}_{\mathrm{mult}}$, each as a C function. These are used in the implementation in C of the masked AES-128 algorithm. I had the mission of coding both implementations during my internship. They are made publicly available[1].
Table 4.1 shows an example of execution of the implemented expanding circuit compiler, on the gadgets instantiated in Section 4.2.1, with different expansion levels k, showing their corresponding gates count vectors, and their execution time as C functions on an Intel i7-8550U CPU. Variables' type is chosen to be the C type **uint8_t**, since these gadgets will also be used for the instantiation of the AES-128 algorithm.

It can be observed that both the complexity and running time grow by almost the same factor

[1] **https://github.com/CryptoExperts/poc-expanding-compiler**

Table 4.1: Complexity and execution time (in ms, on an Intel i7-8550U CPU) for compiled gadgets $G^{(k)}_{\text{add}}$, $G^{(k)}_{\text{copy}}$, $G^{(k)}_{\text{mult}}$ from Section 4.2.1 implemented in C.

k	# shares	Gadget	Complexity $(N_{\text{a}}, N_{\text{c}}, N_{\text{m}}, N_{\text{r}})$	Execution time
1	3	$G^{(1)}_{\text{add}}$	(15, 6, 0, 6)	$1,69.10^{-4}$
		$G^{(1)}_{\text{copy}}$	(12, 9, 0, 6)	$1,67.10^{-4}$
		$G^{(1)}_{\text{mult}}$	(28, 23, 9, 11)	$5,67.10^{-4}$
2	9	$G^{(2)}_{\text{add}}$	(297, 144, 0, 144)	$2,21.10^{-3}$
		$G^{(2)}_{\text{copy}}$	(288, 153, 0, 144)	$2,07.10^{-3}$
		$G^{(2)}_{\text{mult}}$	(948, 582, 81, 438)	$9,91.10^{-3}$
3	27	$G^{(3)}_{\text{add}}$	(6183, 3078, 0, 3078)	$9,29.10^{-2}$
		$G^{(3)}_{\text{copy}}$	(6156, 3105, 0, 3078)	$9,84.10^{-2}$
		$G^{(3)}_{\text{mult}}$	(23472, 12789, 729, 11385)	$3,67.10^{-1}$

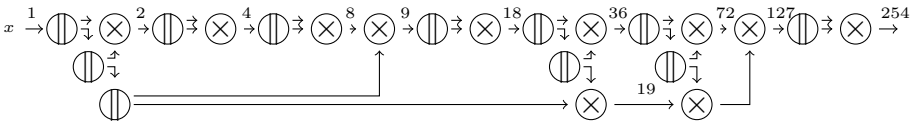

Figure 4.1: Circuit for the exponentiation $x \mapsto x^{254}$.

with the expansion level, with multiplication gadgets being the slowest as expected. Base gadgets with $k = 1$ roughly take 10^{-4} ms, while these gadgets expanded 2 times ($k = 3$) take between 10^{-2} and 10^{-1} ms. The difference between the linear cost of addition and copy gadgets, and the quadratic cost of multiplication gadgets can also be observed through the gadgets' complexities.

Next, an n-share AES implementation in C protected with our instantiation of the expanded gadgets is also provided. The correspoding AES **circuit** performs operations in $\mathbb{K} = GF(256)$. Linear operations in the AES circuit (MixColumns, ShiftRows, AddRoundKey) are considered as in a standard AES, while adding the necessary copy gates. Concerning the non-linear part (sbox computation), for the first part of the sbox which is the exponentiation part ($x \mapsto x^{254}$), we use the circuit representation of the processing proposed in [14] illustrated in figure 4.1. It corresponds to the *addition chain* $(1, 2, 4, 8, 9, 18, 19, 36, 55, 72, 127, 254)$ and it has been chosen due to its optimality regarding the number of multiplications (11 in total). Each time an intermediate result had to be reused, a copy gate (marked with $\|$) has been inserted.
For the second part of the sbox, the affine function is implemented according to the following equation:

$$\text{Affine}(x) = (((((((207x)^2 + 22x)^2 + 1x)^2 + 73x)^2 + 204x)^2 + 168x)^2 + 238x)^2 + 5x + 99$$

Table 4.2: AES operations complexity.

AES Operation	**Complexity** $(N_{\mathbf{a}}, N_{\mathbf{c}}, N_{\mathbf{m}}, N_{\mathbf{r}})$
AddRoundKey (for 1 byte)	$(1, 0, 0, 0)$
SubBytes (for 1 byte)	$(8, 25, 26, 0)$
MixColumns (for all columns)	$(60, 60, 16, 0)$
ShiftRows (for all rows)	$(0, 0, 0, 0)$
AES-128 encryption	$(\mathit{1996}, \mathit{4540}, \mathit{4304}, \mathit{0})$
SubBytes Inversion (for 1 byte)	$(8, 25, 26, 0)$
MixColumns Inversion (for all columns)	$(104, 104, 36, 0)$
ShiftRows Inversion (for all rows)	$(0, 0, 0, 0)$
AES-128 decryption	$(\mathit{2392}, \mathit{4936}, \mathit{4484}, \mathit{0})$

Table 4.3: Standard and n-share AES-128 execution time (in ms, on an Intel i7-8550U CPU) using compiled gadgets $G^{(k)}_{\text{add}}$, $G^{(k)}_{\text{copy}}$, $G^{(k)}_{\text{mult}}$ from Section 4.2.1.

AES Version	**Execution Time (in ms)**	
	Encryption	**Decryption**
Standard (no sharing)	0.06	0.05
3-share ($k = 1$)	1.08	1.07
9-share ($k = 2$)	11.71	10.26
27-share ($k = 3$)	200.29	197.70

with the necessary copy gates. Similarly, the inverse of the affine function is implemented for the sbox inversion as follows:

$$\text{Affine}^{-1}(x) = (((((((147x)^2 + 146x)^2 + 190x)^2 + 41x)^2 + 73x)^2 + 139x)^2 + 79x)^2 + 5x + 5$$

Table 4.2 shows the total gates count vectors for each of the operations in the AES encryption/decryption procedures, as well as their total complexities.
For the n^k-share version of AES, each gate will be replaced by the corresponding compiled gadget $G^{(k)}_{\text{add}}$, $G^{(k)}_{\text{copy}}$, $G^{(k)}_{\text{mult}}$ instantiated in Section 4.2.1. We display in Fig. 4.2 the total number of gates in the AES-128 encryption/decryption procedures as functions of the level k, using the compiled gagdets, and the complexity analysis from Section 4.1.1. For instance, for level $k = 9$, assuming a leakage probability of $p \approx 2^{-8}$ (which is the maximum we can tolerate using our instantiated gadget as computed by **VRAPS**), we achieve a security of $\epsilon \approx 2^{-76}$. At this level, the AES algorithm would count around 10^{16} gates. Table 4.3 shows the AES-128 execution time on a

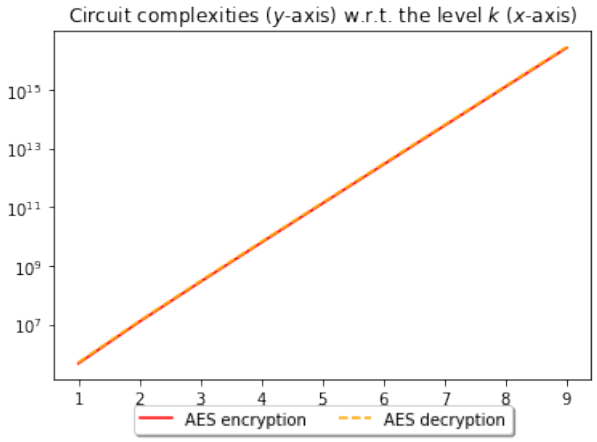

Figure 4.2: Number of gates after compilation of AES-128 encryption/decryption circuits with respect to the level k.

16-byte message with 10 pre-computed sub-keys, using compiled gadgets $G^{(k)}_{\text{add}}$, $G^{(k)}_{\text{copy}}$, $G^{(k)}_{\text{mult}}$, with respect to the expansion level k and sharing order $n = 3^k$. It can be seen that the execution time increases with the expansion level with a similar growth as in Table 4.1. This is because the complexity of the AES circuit strongly depends on the gadgets that are used to replace each gate in the original arithmetic circuit. For example, with our 3-share gadgets that tolerate a leakage probability of $p \approx 2^{-8}$, a 27-share ($k = 3$) AES-128 takes almost 200 milliseconds to encrypt or decrypt a message.

Chapter 5

Further Analysis: Amplification Order & Complexity

During the rest of my internship, and after all of the work related to our published paper was done, I started concentrating more on the expansion strategy and its complexity. In this chapter, I will present the analysis I made this far related to achievable amplification orders of the failure function f for RPE property. Then, I will use this to study the complexity of the expansion strategy. This chapter contains a first analysis, which will be used when I start my thesis at CryptoExperts in October 2020, to establish more sophisticated and interesting results on the subject.

5.1 Amplification Order

As explained in section 4.1.1, the failure event function f for (t, f)-RPE is of the form

$$f(p) = c_d p^d + \mathcal{O}(p^{d+1})$$

of **amplification order** d. The amplification order of the function is the parameter that determines the number of expansion levels that have to be done in order to reach a certain security level κ. This expresses as a complexity in κ for an expanded circuit $\hat{C}$

$$|\hat{C}| = \mathcal{O}(|C|.\kappa^e) \quad , \quad with \;\; e = \frac{\log(N_{max})}{\log(d)}$$

Meanwhile, an expanding circuit compiler of sharing order n and base n-share gadgets G_{add}, G_{copy}, G_{mult}, cannot achieve an arbitrary desired amplification order. In fact, there is a bound on the amplification order for $f(p)$ in (t, f)-RPE for n-share gadgets.

Lemma 1. Let G be an n-share (t, f)-RPE gadget. Then the amplification order d of f is bounded

by:

1. $d \leq min((t+1), 2(n-t))$ if G has only 1 input.

2. $d \leq min((t+1), (n-t))$ if G has 2 inputs.

Proof. Consider a gadget G which is $(t,\ f)$-RPE. The first bound on the amplification order $d \leq (t+1)$ is direct for both points 1 and 2, since by probing t+1 shares of any input, the set W that contains them will be a failure set (since $t+1 > t$), and so the function f for RPE will have a non-zero coefficient in p^{t+1}. For the rest of the proof, we will prove the amplification order bound on f^1 in the (t, f^1)-RPE1 property (see Section 3.1.3). And since f for RPE is the maximum between the functions f^1 for RPE1 and f^2 for RPE2 ($f = \max(f^1, f^2)$), then the bound on the amplification order will also apply on f for (t, f)-RPE property.

1. *Proof for the bound* $2(n-t)$ *:* Let G be a gadget with 1 input. We will exhibit a failure set W of size $2(n-t)$ with t output shares, so that the function f^1 for RPE1 has a non-zero coefficient in $p^{2(n-t)}$. Consider the output shares $z_1, \ldots, z_n$ for output z (for 2 outputs gadgets, we can reason on only 1 output). For the RPE1 property, t shares of z are considered in each tested tuple. Without loss of generality, let $z_1, \ldots, z_t$ be those shares. Each output share z_i is in fact an output wire of a gate. Let us probe both input wires of each of these gates for output shares $z_{t+1}, \ldots, z_n$, giving us a set W of size $2(n-t)$. The resulting set $W + \{z_1, \ldots, z_t\}$ then contains the full output z and so requires the knowledge of the full input to be simulated. This means that we need all $n > t$ shares of the input for simulation. This means that this set is a failure set and contains $2(n-t)$ probes, so the function f^1 for RPE1 will have a non-zero coefficient in $p^{2(n-t)}$, so the amplification order d of f^1 is $d \leq 2(n-t)$.

2. *Proof for the bound* $(n-t)$ *:* Let G be a gadget with 2 inputs. Considering the same failure set as in the above case, the knowledge of the full output means that we need both full inputs to simulate the set (since the full output requires the full input), so f^1_{12} for RPE1 (failure on both inputs) will have a non-zero coefficient in $p^{2(n-t)}$. Since for 2 inputs gadgets the function f^1 for RPE1 is the maximum of f^1_1, f^1_2 and $\sqrt{f^{12}_1}$, then the amplification order of f^1 is bounded by $d \leq (n-t)$ ($\sqrt{p^{2(n-t)}} = p^{n-t}$)

By proving both bounds, we conclude that $d \leq min((t+1), 2(n-t))$ for 1 input gadgets, and $d \leq min((t+1), (n-t))$ for 2 inputs gadgets. □

This implies that the maximum amplification order achievable for n-share 1 input gadgets is when

$$d = t + 1 = 2(n-t)$$

So,

$$t = \frac{2n-1}{3}$$

giving us

$$d_{\max} = \frac{2(n+1)}{3}$$

Using the same reasoning, the maximum amplification order achievable for 2 inputs gadgets is when $t = \frac{n-1}{2}$, and so

$$d_{\max} = \frac{n+1}{2}$$

When instantiating an expanding circuit compiler CC with base n-share gadgets G_{add}, G_{copy} and G_{mult}, the failure event function f_{CC} associated to the compiler will be the maximum of all the functions of the gadgets

$$f_{CC} = \max(f_{G_{\text{add}}}, \quad f_{G_{\text{copy}}}, \quad f_{G_{\text{mult}}}) \tag{5.1}$$

And since G_{add} and G_{mult} are 2 inputs gadgets, then the amplification order associated to the compiler CC and to the final asymptotic complexity estimation has the same bounds as the amplification order for 2 inputs gadgets. Namely, the maximum amplification order $d_{\max}$ achievable by any compiled circuit $\hat{C}$ with the expanding circuit compiler is

$$d_{\max} = \frac{n+1}{2} \tag{5.2}$$

when $t = \frac{n-1}{2}$.
It is clear that the higher the sharing order n, the higher the maximum amplification order that we can achieve.

5.2 $N_{\max}$

Recall from Section 4.1.1 that the complexity of a compiled circuit with the expanding circuit compiler is in $\mathcal{O}(|C|.\kappa^e)$ for a base circuit C, a security parameter κ, and with $e = \frac{\log(N_{\max})}{\log(d)}$, where $N_{\max}$ is the maximum of the eigen values of the gates count matrix M defined in (4.1).
Addition and copy gadgets often use linear operations and no multiplication gates. If that is the case, then $N_{add,m} = N_{copy,m} = 0$. It can be checked in this case that the eigenvalues of the matrix

M are

$$\lambda_1, \lambda_2 = \mathsf{eigenvalues}(M_{ac}) \quad , \quad \lambda_3 = N_{mult,m} \quad , \quad \lambda_4 = n$$

where M_{ac} is the upper sub-matrix

$$M_{ac} = \begin{pmatrix} N_{add,a} & N_{copy,a} \\ N_{add,c} & N_{copy,c} \end{pmatrix}$$

So, we can simply say that $N_{\max} = \max\Big(\mathsf{eigenvalues}(M_{ac}), N_{mult,m}\Big)$. Informally, this means that the value of $N_{\max}$ depends on the number of multiplication gates in the gadget G_{mult}, and the copy and add gates in G_{add} and G_{copy}.
For example, if G_{mult} has $\mathcal{O}(n^2)$ multiplication gates (which is the basic multiplication algorithm cost), and considering that this cost is bigger than λ_1, λ_2, then $N_{\max} = \mathcal{O}(n^2)$.

5.3 Asymptotic Complexity

Figure 5.1 shows the variation of the complexity exponent e from (4.4) with respect to the maximum amplification order achievable for an expanding circuit compiler $d_{\max}$, and the value of $N_{\max}$ described earlier. Several functions have been chosen for the value of $N_{\max}$ ($\mathcal{O}(n^2)$, $\mathcal{O}(n^{1.5})$, $\mathcal{O}(n\log(n))$, $\mathcal{O}(n)$), especially when the cost of the multiplication gadget G_{mult} can be reduced with more sophisticated multiplication circuits, and when it is dominant over the cost of the addition and copy gadgets G_{add}, G_{copy}. The value of n is chosen as the minimal number of shares necessary to achieve the considered maximum amplification order $d_{\max}$. Namely, from (5.2), we get that

$$n_{\min} = 2d_{\max} - 1$$

It can be seen that the value of the exponent e decreases as the amplification order and $N_{\max}$ increase. Since $e = \dfrac{\log(N_{\max})}{\log(d)}$, this means that d increases slightly more rapidly than $N_{\max}$. When $N_{\max} = \mathcal{O}(n^2)$ (when we use $\mathcal{O}(n^2)$ multiplication gates in G_{mult}), the value of e decreases from 3.25 with less than 5-share gadgets, to almost 2.5 with almost 13-share gadgets. Note that using gadgets with more than 20 to 25 shares is not very interesting in practice.
As the value of $N_{\max}$ decreases, the value of e decreases too. For example, if G_{mult} uses $\mathcal{O}(n\log(n))$ multiplication gates, and dominates the cost of G_{add} and G_{copy}, then we can achieve $e_{\min} \approx 1.7$ on average.
Compared to the complexity obtained with our instantiation, when we use $\mathcal{O}(n^2)$ multiplication

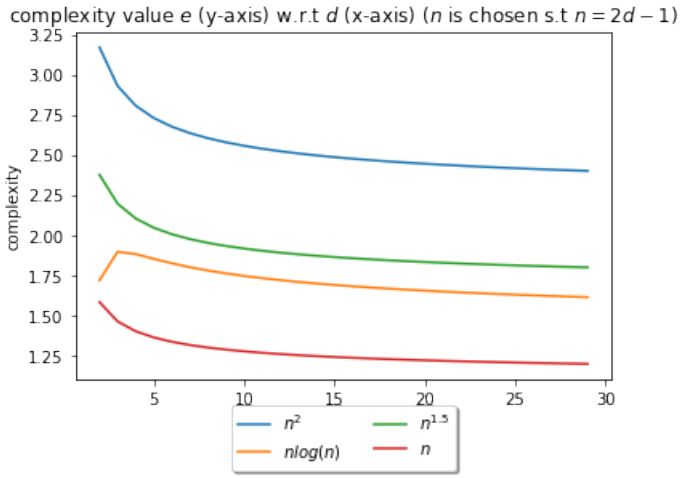

Figure 5.1: Exponent e with respect to $N_{\max}$ and maximum amplification order $d_{\max} = \dfrac{n+1}{2}$.

gates in G_{mult}, recall that we have

$$|\hat{C}| = \mathcal{O}(|C|.\kappa^{7.5})$$

with 3-share base gadgets. So it is clear that there is still much room for optimization of our construction. In addition, we do not achieve the bound on the amplification order for 3-share gadgets stated in Lemma 1.

This gives us a motivation to look for other constructions that can achieve this bound, and maybe find generic constructions for which we can prove that they attain the maximum bound on the amplification order for any number of shares n, rather than a fixed one. This will be one of the first ideas that I will be working on during the beginning of my thesis at CryptoExperts.

Chapter 6

Conclusion

To conclude, my internship's main subject was the theoretical security of masking schemes against side-channel attacks. In our work, we proved this security in the "closer-to-reality" leakage model which is the random probing model.

Our work is one of the first frameworks dedicated to the random probing model. We were able to formally define security in the random probing model, and provide a formal verification tool called **VRAPS** which computes the security parameters for any small circuit in this model. I accomplished my mission of providing a full optimized implementation of this tool which was made publicly available by CryptoExperts company, for users to test on any small circuit they create.

We also introduced new notions of composition and expansion for achieving global arbitrary random probing security levels. Our expansion strategy is inspired from the AIS work based on multiparty computation protocols, and is easier and more practical to instantiate than the latter. I accomplished my other mission of providing a detailed analysis of the AIS compiler complexity and compared it to our strategy's complexity. Concretely, we managed to instantiate our strategy with 3-share base gadgets and showed using our tool, that we could achieve a complexity in $\mathcal{O}(|C|.\kappa^{7.5})$ while tolerating a leakage probability of $p_{\max} \approx 2^{-8}$. This was rigorously compared to the complexity of the AIS instantiation in $\mathcal{O}(|C|.\kappa^{8.19})$ and which tolerates a much lower leakage probability of $p_{\max} \approx 2^{-26}$. I additionally accomplished my mission of providing a public full implementation of our expansion strategy that operates on small circuits, as well as an implementation in C language of the AES algorithm that uses expanded gadgets, as a concrete example of usage of this procedure.

Finally, I provided a first analysis for future works during my phD, which will aim to look for generic constructions that achieve the bounds on amplification orders for any sharing order n. We will also be looking more into complexity-tolerated probability trade offs that can be offered with these constructions.

Bibliography

[1] M. Ajtai. Secure computation with information leaking to an adversary. In L. Fortnow and S. P. Vadhan, editors, *43rd Annual ACM Symposium on Theory of Computing*, pages 715–724, San Jose, CA, USA, June 6–8, 2011. ACM Press.

[2] P. Ananth, Y. Ishai, and A. Sahai. Private circuits: A modular approach. In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 427–455, Santa Barbara, CA, USA, Aug. 19–23, 2018. Springer, Heidelberg, Germany.

[3] M. Andrychowicz, S. Dziembowski, and S. Faust. Circuit compilers with $O(1/\log(n))$ leakage rate. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 586–615, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[4] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub. Verified proofs of higher-order masking. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485, Sofia, Bulgaria, Apr. 26–30, 2015. Springer, Heidelberg, Germany.

[5] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, P.-Y. Strub, and R. Zucchini. Strong non-interference and type-directed higher-order masking. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 116–129, Vienna, Austria, Oct. 24–28, 2016. ACM Press.

[6] G. Barthe, F. Dupressoir, S. Faust, B. Grégoire, F.-X. Standaert, and P.-Y. Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In J. Coron and J. B. Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, Paris, France, Apr. 30 – May 4, 2017. Springer, Heidelberg, Germany.

[7] A. Battistello, J.-S. Coron, E. Prouff, and R. Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In B. Gierlichs and A. Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, volume 9813 of *Lecture Notes in Computer Science*, pages 23–39, Santa Barbara, CA, USA, Aug. 17–19, 2016. Springer, Heidelberg, Germany.

[8] S. Belaïd, F. Benhamouda, A. Passelègue, E. Prouff, A. Thillard, and D. Vergnaud. Randomness complexity of private circuits for multiplication. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 616–648, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[9] S. Belaıd, J.-S. Coron, E. Prouff, M. Rivain, and A. R. Taleb. Random probing security: Verification, composition, expansion and new constructions.

[10] S. Belaïd, D. Goudarzi, and M. Rivain. Tight private circuits: Achieving probing security with the least refreshing. In T. Peyrin and S. Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 343–372, Brisbane, Queensland, Australia, Dec. 2–6, 2018. Springer, Heidelberg, Germany.

[11] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412, Santa Barbara, CA, USA, Aug. 15–19, 1999. Springer, Heidelberg, Germany.

[12] J.-S. Coron. Formal verification of side-channel countermeasures via elementary circuit transformations. In B. Preneel and F. Vercauteren, editors, *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*, volume 10892 of *Lecture Notes in Computer Science*, pages 65–82, Leuven, Belgium, July 2–4, 2018. Springer, Heidelberg, Germany.

[13] J.-S. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-order side channel security and mask refreshing. In S. Moriai, editor, *Fast Software Encryption – FSE 2013*, volume 8424 of *Lecture Notes in Computer Science*, pages 410–424, Singapore, Mar. 11–13, 2014. Springer, Heidelberg, Germany.

[14] I. Damgård and M. Keller. Secure multiparty AES. In R. Sion, editor, *FC 2010: 14th International Conference on Financial Cryptography and Data Security*, volume 6052 of *Lecture*

Notes in Computer Science, pages 367–374, Tenerife, Canary Islands, Spain, Jan. 25–28, 2010. Springer, Heidelberg, Germany.

[15] A. Duc, S. Dziembowski, and S. Faust. Unifying leakage models: From probing attacks to noisy leakage. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 423–440, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.

[16] L. Goubin and J. Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES'99*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172, Worcester, Massachusetts, USA, Aug. 12–13, 1999. Springer, Heidelberg, Germany.

[17] Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In D. Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481, Santa Barbara, CA, USA, Aug. 17–21, 2003. Springer, Heidelberg, Germany.

[18] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397, Santa Barbara, CA, USA, Aug. 15–19, 1999. Springer, Heidelberg, Germany.

[19] U. M. Maurer. Secure multi-party computation made simple (invited talk). In S. Cimato, C. Galdi, and G. Persiano, editors, *SCN 02: 3rd International Conference on Security in Communication Networks*, volume 2576 of *Lecture Notes in Computer Science*, pages 14–28, Amalfi, Italy, Sept. 12–13, 2003. Springer, Heidelberg, Germany.

[20] E. Prouff and M. Rivain. Masking against side-channel attacks: A formal security proof. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[21] M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427, Santa Barbara, CA, USA, Aug. 17–20, 2010. Springer, Heidelberg, Germany.

Appendices

Appendix A

RPE Verification Algorithms

In this appendix, I give the full algorithms for the RPE verification procedure described in section 3.1.3. Algorithm 3 computes the function f^1 for the (t, f^1)-RPE1 property. Algorithm 4 computes the function f^2 for the (t, f^2)-RPE2 property. And Algorithm 5 computes the function $f = \max(f^1, f^2)$ for the (t, f)-RPE property.

The functions used in these algorithms are the same functions used in Algorithms 1 and 2, except for the function failureTest3. This function applies the set of rules for the search of failure tuples, and considers a simulation failure when $t + 1$ shares of the secret inputs are needed for the simulation of W and J, similar to the function failureTest2. The difference from the latter is that it splits failure sets into failures on each input. Output list $L1_p$ contains sets W for which the simulation of W and J requires the knowledge of $t + 1$ or more shares of the first input. And output list $L2_p$ contains sets W for which the simulation of W and J requires the knowledge of $t + 1$ or more shares of the second input. The list of failure sets on both inputs is obtained by the intersection of both output lists $L1_p \cap L2_p$.

Note that in Algorithm 4, there is a double loop. The outer loop iterates over the sizes of sets W for the coefficients c_1 through c_β. Then, the inner loop iterates over all sets of output share indices J of size $n - 1$. The aim of this loop is to look for sets W for which there is a simulation failure with all such sets J. Which is why at each iteration, on line 10, an intersection of the newly found failure sets W and the previous failure sets is done ($L1_p \leftarrow L1_p \cap L1'_p$ and $L2_p \leftarrow L2_p \cap L2'_p$). At the the end of this loop, failure sets will only be considered if they resulted in a simulation failure with all possible sets of output shares J. This means that for such sets, there is no set of $n - 1$ output shares J for which the simulation succeeds, which does not fulfill the condition from definition 9, resulting in a failure set for RPE2.

Algorithm 3 VRAPS: RPE **(1)** Security Verification

Input: a 2-to-1 n-share gadget G with s wires, a threshold $\beta \leq s$ and a parameter $1 \leq t < n$.
Output: a lists of β coefficients $c_1^{(1)}$, ..., $c_\beta^{(1)}$ for f_1, $c_1^{(2)}$, ..., $c_\beta^{(2)}$ for f_2 and $c_1^{(12)}$, ..., $c_\beta^{(12)}$ for f_{12}

$(c_1^{(1)}, \ldots, c_\beta^{(1)}), (c_1^{(2)}, \ldots, c_\beta^{(2)}), (c_1^{(12)}, \ldots, c_\beta^{(12)}) \leftarrow (0, \ldots, 0), (0, \ldots, 0), (0, \ldots, 0)$
$L_J \leftarrow \mathsf{listOutputTuples}(n,t)$ ▷ list of all possible sets J of t output shares labels

for J in L_J **do**
    $(c_1'^{(1)}, \ldots, c_\beta'^{(1)}), (c_1'^{(2)}, \ldots, c_\beta'^{(2)}), (c_1'^{(12)}, \ldots, c_\beta'^{(12)}) \leftarrow (0, \ldots, 0), (0, \ldots, 0), (0, \ldots, 0)$

    for $i = 1$ to β **do**
        $L_{W_i} \leftarrow \mathsf{listTuples}(s,i)$ ▷ list of $\binom{s}{i}$ possible sets W of i wire labels
        $L1_p, L2_p \leftarrow \mathsf{failureTest3}(G, L_{W_i}, J)$ ▷ identify failure tuples in L_{W_i} on each of the inputs and store them in $L1_p$ and $L2_p$ respectively

        $c_i'^{(1)} \leftarrow \mathsf{SizeOf}(L1_p)$, $c_i'^{(2)} \leftarrow \mathsf{SizeOf}(L2_p)$, $c_i'^{(12)} \leftarrow \mathsf{SizeOf}(L1_p \cap L2_p)$
    end for

    for $i = 1$ to β **do**
        $c_i^{(1)} \leftarrow \max(c_i^{(1)}, c_i'^{(1)})$, $c_i^{(2)} \leftarrow \max(c_i^{(2)}, c_i'^{(2)})$, $c_i^{(12)} \leftarrow \max(c_i^{(12)}, c_i'^{(12)})$
    end for
end for
return $(c_1^{(1)}, \ldots, c_\beta^{(1)})$, $(c_1^{(2)}, \ldots, c_\beta^{(2)})$, $(c_1^{(12)}, \ldots, c_\beta^{(12)})$

Algorithm 4 VRAPS: RPE **(2)** Security Verification

Input: a 2-to-1 n-share gadget G with s wires, a threshold $\beta \leq s$ and a parameter $1 \leq t < n$.
Output: a lists of β coefficients $c_1^{(1)}$, ..., $c_\beta^{(1)}$ for f_1, $c_1^{(2)}$, ..., $c_\beta^{(2)}$ for f_2 and $c_1^{(12)}$, ..., $c_\beta^{(12)}$ for f_{12}

$(c_1^{(1)}, \ldots, c_\beta^{(1)}), (c_1^{(2)}, \ldots, c_\beta^{(2)}), (c_1^{(12)}, \ldots, c_\beta^{(12)}) \leftarrow (0, \ldots, 0), (0, \ldots, 0), (0, \ldots, 0)$
$L_J \leftarrow \mathsf{listOutputTuples}(n, n-1)$ ▷ list of all possible sets J of $n-1$ output shares labels

for $i = 1$ to β **do**
  $L_{W_i} \leftarrow \mathsf{listTuples}(s, i)$ ▷ list of $\binom{s}{i}$ possible sets W of i wire labels
  $L1_p, L2_p \leftarrow \mathsf{failureTest3}(G, L_{W_i}, L_J[0])$ ▷ identify failure tuples in L_{W_i} on each of the inputs and store them in $L1_p$ and $L2_p$ respectively

  for J in $L_J[1:]$ **do**
    $L1'_p, L2'_p \leftarrow \mathsf{failureTest3}(G, L_{W_i}, J)$
    $L1_p \leftarrow L1_p \cap L1'_p \quad , \quad L2_p \leftarrow L1_p \cap L2'_p$
  end for

  $c_i^{(1)} \leftarrow \mathsf{SizeOf}(L1_p), \; c_i^{(2)} \leftarrow \mathsf{SizeOf}(L2_p), \; c_i^{(12)} \leftarrow \mathsf{SizeOf}(L1_p \cap L2_p)$
end for
return $(c_1^{(1)}, \ldots, c_\beta^{(1)}), \; (c_1^{(2)}, \ldots, c_\beta^{(2)}), \; (c_1^{(12)}, \ldots, c_\beta^{(12)})$

Algorithm 5 VRAPS: RPE **(1 & 2)** Security Verification

Input: a 2-to-1 n-share gadget G with s wires, a threshold $\beta \leq s$ and a parameter $1 \leq t < n$.
Output: a lists of β coefficients c_1, ..., c_β for f

$(c_1^{(1)}, \ldots, c_\beta^{(1)}), \; (c_1^{(2)}, \ldots, c_\beta^{(2)}), \; (c_1^{(12)}, \ldots, c_\beta^{(12)}) \leftarrow \mathsf{Algorithm3}(G, \beta, t)$ ▷ RPE1 Verification

$(C_1^{(1)}, \ldots, C_\beta^{(1)}), \; (C_1^{(2)}, \ldots, C_\beta^{(2)}), \; (C_1^{(12)}, \ldots, C_\beta^{(12)}) \leftarrow \mathsf{Algorithm4}(G, \beta, t)$
▷ RPE2 Verification

$(c_1, \ldots, c_\beta) \leftarrow (0, \ldots, 0)$
for $i = 1$ to β **do**
  $c_i \leftarrow \max\left(c_i^{(1)}, C_i^{(1)}, c_i^{(2)}, C_i^{(2)}, \sqrt{c_i^{(12)}}, \sqrt{C_i^{(12)}}\right)$
end for
return $(c_1, \ldots, c_\beta)$

Appendix B

RPE Instantiation

In this appendinx, I will first prove the result mentioned in section 4.2.1 regarding 2-share gadgets. Mainly, there are no (2-share, 2-to-1) $(1, f)$-RPE gadgets such that f has an amplification order greater than one.
Let G be a 2-share gadgets with output sharing $z = (z_0, z_1)$ and inputs sharings $x = (x_0, x_1)$, $y = (y_0, y_1)$. For G to be $(1, f)$-RPE with f of amplification order strictly greater than one, it should be (t, f^1)-RPE1 with f^1 of order strictly greater than one (since $f = \max(f^1, f^2)$). Specifically, f^1_{12} must be of amplification strictly greater than two (since $f^1 = \max(f^1_1, f^1_2, \sqrt{f^1_{12}})$). In other words, we should be able to exhibit a simulator such that one share of each input is enough to simulate anyone of the output shares and an arbitrary couple of leaking wires. But the output wire z_0 and both input gates of the second output share z_1 represent the full output and require the knowledge of both inputs to be simulated. Therefore, f^1_{12} has a non-zero coefficient in p^2 and is thus not of amplification order strictly greater than two. Then f^1 is not of amplification order strictly greater than one, and neither is f in $(1, f)$-RPE.
For this reason, we choose to construct gadgets with sharing order $n \geq 3$. The following gadgets are 3-share gadgets that were used to instantiate our expanding compiler in section 4.2.1.

In the following gadgets, variables r_i are fresh random values, operations are processed with the usual priority rules, and the number of implicit copy gates can be deduced from the occurrences of each intermediate variable such that n occurrences require $n - 1$ implicit copy gates.
The 3-share G_{add} gadget is based on a circular refreshing scheme as introduced in [6], wile rearranging the order of the random values.

$$\begin{aligned} G_{\text{add}} : z_0 &\leftarrow x_0 + r_0 + r_4 + y_0 + r_1 + r_3 \\ z_1 &\leftarrow x_1 + r_1 + r_5 + y_1 + r_2 + r_4 \\ z_2 &\leftarrow x_2 + r_2 + r_3 + y_2 + r_0 + r_5 \end{aligned}$$

where x and y are the input sharings, and z the output sharing. The copy gadget G_{copy} construction relies on the same circular refreshing gadget.

$$\begin{aligned} G_{\text{copy}} : v_0 &\leftarrow u_0 + r_0 + r_1;\ w_0 \leftarrow u_0 + r_3 + r_4 \\ v_1 &\leftarrow u_1 + r_1 + r_2;\ w_1 \leftarrow u_1 + r_4 + r_5 \\ v_2 &\leftarrow u_2 + r_2 + r_0;\ w_2 \leftarrow u_2 + r_5 + r_3 \end{aligned}$$

with input u, and outputs v and w.
Finally, the gadget G_{mult} first refreshes both inputs, before any multiplication is performed

$$\begin{aligned} G_{\text{mult}} : u_0 &\leftarrow x_0 + r_5 + r_6; \qquad u_1 \leftarrow x_1 + r_6 + r_7; \qquad u_2 \leftarrow x_2 + r_7 + r_5 \\ v_0 &\leftarrow y_0 + r_8 + r_9; \qquad v_1 \leftarrow y_1 + r_9 + r_{10}; \qquad v_2 \leftarrow y_2 + r_{10} + r_8 \\ z_0 &\leftarrow \Big(u_0 \cdot v_0 + r_0\Big) + \Big(u_0 \cdot v_1 + r_1\Big) + \Big(u_0 \cdot v_2 + r_2\Big) \\ z_1 &\leftarrow \Big(u_1 \cdot v_0 + r_1\Big) + \Big(u_1 \cdot v_1 + r_4\Big) + \Big(u_1 \cdot v_2 + r_3\Big) \\ z_2 &\leftarrow \Big(u_2 \cdot v_0 + r_2\Big) + \Big(u_2 \cdot v_1 + r_3\Big) + \Big(u_2 \cdot v_2 + r_0\Big) + r_4 \end{aligned}$$

with inputs x and y, and output z.
Table B.1 shows the functin $f(p)$ corresponding to each of the above gadgets as computed by our verification tool **VRAPS** using Algorithm 5 with $t = 1$. The final function associated to the expanding compiler CC is the maximum of all the computed functions (i.e the one with the smallest amplification order), giving us the function $f_{\max} = \sqrt{83}p^{3/2} + \mathcal{O}(p^2)$ with an amplification order of $\dfrac{3}{2}$.

Table B.1: Functions $f(p)$ for each of the 3-share constructed gadgets G_{add}, G_{copy} and G_{mult} for $(t = 1, f)$-RPE as computed by the verification tool **VRAPS**

Gadget	Complexity $(N_{\text{a}}, N_{\text{c}}, N_{\text{m}}, N_{\text{r}})$	f in $(t = 1, f)$-RPE computed by **VRAPS**
G_{add}	$(15, 6, 0, 6)$	$\sqrt{69}p^2 + \mathcal{O}(p^3)$
G_{copy}	$(12, 9, 0, 6)$	$33p^2 + \mathcal{O}(p^3)$
G_{mult}	$(28, 23, 9, 11)$	$\sqrt{83}p^{3/2} + \mathcal{O}(p^2)$

Appendix C

AIS Instantiation with Maurer (m, c)-MPC protocol [19]

In this appendix, I provide a full analysis of the Maurer multiparty computation (m, c)-MPC protocol [19] of m parties tolerating c corruptions, which is used for the instantiation of the AIS compiler [2] exhibited in section 4.2.2.
First, using this compiler, given a circuit C to compile, each gate G is implemented using a functionality F associated to the MPC protocol Π. Such a functionality F receives m shares of each input and then reconstructs them to obtain original values. This reconstruction can be done with $2(m-1)$ addition gates. Then after computing the gate G, m additive shares of the output are computed twice. This step consists of one gate for G, and $2(m-1)$ gates for the additive sharing along with $2(m-1)$ random gates.[1] So each gate G to compile is replaced by $6m-5$ gates, each computed jointly by the m parties in the MPC protocol. Next, I state the complexity of the protocol from [19]. Each gate in a functionality F is jointly computed by all m parties. In the beginning, each party holds one share of each input.

The first step consists in a k-secret sharing of each input share where $k = \binom{m}{c}$. For an input of m shares, each party will hold a total of $m\binom{m-1}{c}$ shares. For two inputs, this step has a complexity of $m(2k-2)$.

The second step is either performing an addition or a multiplication, depending on the gate G associated to the functionality. An addition simply means each party locally adding all its shares, holding a complexity of $m\binom{m-1}{c}$. In case of a multiplication gate, each party will locally compute the sum of the product of the shares of both inputs, and then share its local result using a secret sharing scheme as in the first step. This procedure holds a complexity of $\binom{m-1}{c}^2$ for computing the result, $m(2k-2)$ for the secret sharing, and $2k^2$ copy gates. Clearly, the cost of the second

[1] In [2], the authors only consider $2(m-1)$ for the cost of this step, not counting the number of random gates necessary to compute the additive sharing of the output.

step is more important for the multiplication and can be upper bounded by[2]

$$\binom{m-1}{c}^2 + m \cdot (2k-2) + 2k^2.$$

In the final step, every party broadcasts its shares to all other parties, and then adds all the shares it receives. The complexity of this step is $\binom{m}{c}$.

Considering the cost of replacing each gate G in the circuit to compile by $6m-5$ gates, and the cost to compute each of these gates using the protocol Π, the total number of gates N_{g} is upper bounded by

$$(6m-5) \cdot \left(\binom{m-1}{c}^2 + m(2k-2) + 2k^2 \right).$$

[2]The authors claim in their paper a complexity of $\binom{m-1}{c}^2 + 2mk$, since they do not take into account the copy gates needed to compute the product of input shares.