PhD-FSTC-2015-23
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defense held on 29/05/2015 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Praveen Kumar Vadnala
Born on 28 July 1983 in Mancherial (India)

# Provably Secure Countermeasures against Side-channel Attacks

## Dissertation defense committee

# Abstract

Side-channel attacks exploit the fact that the implementations of cryptographic algorithms leak information about the secret key. In power analysis attacks, the observable leakage is the power consumption of the device, which is dependent on the processed data and the performed operations. Masking is a widely used countermeasure to thwart the powerful Differential Power Analysis (DPA) attacks. It uses random variables called masks to reduce the correlation between the secret key and the obtained leakage. The advantage with masking countermeasure is that one can formally prove its security under reasonable assumptions on the device leakage model. This thesis proposes several new masking schemes along with the analysis and improvement of few existing masking schemes.

The first part of the thesis addresses the problem of converting between Boolean and arithmetic masking. To protect a cryptographic algorithm which contains a mixture of Boolean and arithmetic operations, one uses both Boolean and arithmetic masking. Consequently, these masks need to be converted between the two forms based on the sequence of operations. The existing conversion schemes are secure against first-order DPA attacks only. This thesis proposes first solution to switch between Boolean and arithmetic masking that is secure against attacks of any order. Secondly, new solutions are proposed for first-order secure conversion with logarithmic complexity ($\mathcal{O}(\log k)$ for $k$-bit operands) compared to the existing solutions with linear complexity ($\mathcal{O}(k)$). It is shown that this new technique also improves the complexity of the higher-order conversion algorithms from $\mathcal{O}(n^2k)$ to $\mathcal{O}(n^2 \log k)$ secure against attacks of order $d$, where $n = 2d+1$. Thirdly, for the special case of second-order masking, the running times of the algorithms are further improved by employing lookup tables.

The second part of the thesis analyzes the security of two existing Boolean masking schemes. Firstly, it is shown that a higher-order masking scheme claimed to be secure against attacks of order $d$ can be broken with an attack of order $d/2+1$. An improved scheme is proposed to fix the flaw. Secondly, a new issue concerning the problem of converting the security proofs from one leakage model to another is examined. It is shown that a second-order masking scheme secure in the Hamming weight model can be broken with a first-order attack on a device leaking in the Hamming distance model. This result underlines the importance of re-evaluating the security proofs for devices leaking in different models.

Dedicated to my family.

# Acknowledgments

First of all, I would like to thank my supervisor Jean-Sébastien Coron for accepting me as his student and guiding me actively throughout my PhD. He gave me ample freedom to purse my ideas and provided feedback at every step on the way. He has been a source of inspiration for many with his world class research and I feel extremely lucky to have an opportunity to work with him.

Secondly, I would like to thank my co-supervisor Johann Großschädl who has given me enormous support throughout my PhD. We have written several papers together and worked very closely. His expertise in efficient implementations has greatly helped us in implementing our ideas and improving the results. He was also part of my doctoral committee and gave me important feedback to improve my research.

I would like to thank Alex Biryukov for serving in my doctoral committee and providing me with useful suggestions. Though we did not work together, it was a great pleasure to know a researcher of his stature personally. Being the head of LACS for major part of my PhD, he allowed me to travel to several conferences/workshops, which has greatly helped my research and provided me with important professional contacts.

I would like to thank Ilya Kizhvatov for helping me with the basic understanding of side-channel attacks, which was a totally new area for me. The measurement setup he built during his PhD has saved me a lot of effort and time. He also guided me during my internship at Riscure and provided me with excellent support. I would also like to thank him for accepting to be part of my defense committee as an expert in advisory capacity.

I would like to thank Emmannuel Prouff, Louis Goubin and François-Xavier Standaert for accepting to be part of my defense committee and providing important feedback in improving this manuscript. I also would like to thank Franck Leprévost for accepting to be the chair of the committee.

A special thanks goes to my Masters thesis supervisor Anish Mathuria who introduced me to the filed of cryptography and made me passionate about the subject. I would like to thank my co-authors Jean-François Gallais, Arnab Roy, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, David Galindo, Zhe Liu, Srinivas Vivek, Mehdi Tibouchi, Junwei Wang and Qiuliang Xu for their collaboration and contribution.

I would like to thank the past and present members of LACS with whom I enjoyed working together. It was a pleasure to be a part of the team involving

Zhe Liu, Srinivas Vivek Venkatesh, Ivan Pustogarov, Vesselin Velichkov, Arnab Roy, Avradip Mandal, Jean-François Gallais, David Galindo, Léo Perrin, Patrick Derbez, Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Gaëtan Leurent, Ralf-Philipp Weinmann, Aleksei Udovenko, Vikash Kumar and Deike Priemuth-Schmid. I would like to thank Andre Stemper who helped me with the practical experiments. A special thanks also goes to the LACS secretaries Fabienne Schmitz and Isabelle Schroeder who helped me with the administrative tasks and travel arrangements. A big thanks to the University of Luxembourg for providing me with generous budget for conference trips and excellent working conditions.

I would like to thank all my friends who made my stay in Luxembourg unforgettable. The last but not the least, a huge thanks to my family for their support and unconditional love. None of this would have been possible without them.

Praveen Kumar Vadnala, May 2015.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

**Contents**

## 1.1 Cryptography

Cryptography is the art and science of securing communications in the presence of a malicious third party. It includes protocols, algorithms and techniques to prevent unauthorized access to the sensitive information. Cryptanalysis on the other hand is the art and science of breaking cryptographic systems. The combination of cryptography and cryptanalysis is called cryptology. Cryptography is used in variety of applications including Internet, ATM, smart cards, passwords and e-commerce.

Suppose Alice (sender) wants to send a message (called *plaintext*) to Bob (receiver). Also suppose that Alice does not want any eavesdropper (Eve) to be able to read the message. This can be achieved by using the techniques provided by cryptography: Encryption and Decryption. Alice encrypts the plaintext using the encryption algorithm and produces a random looking message called *ciphertext*. Then she sends the ciphertext to Bob over an insecure channel (*e.g.* internet) which can be observed by Eve. Upon receiving the message, Bob uses decryption algorithm to decrypt the ciphertext back to the plaintext. This procedure is shown pictorially in Figure 1.1. The encryption algorithm ensures that even if Eve gets hold of the ciphertext, she will not be able to recover the plaintext from it. The combination of encryption and decryption algorithms are referred to as *cipher*.

The use of encryption for secure communication has been around for many centuries. The classical ciphers generally fall into two categories: *transposition cipher* and *substitution cipher*. The transposition ciphers rearrange the letters of a message, whereas substitution ciphers replace one or a group of letters with other letter

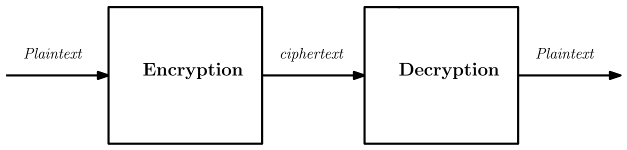


Figure 1.1: The process of encryption and decryption

or a group of letters. For example, in Caeser cipher all the letters in the plaintext are shifted by some fixed number of times (called *key*) to produce the cipher text. The Vigénere cipher extends this idea by using different Caeser ciphers which in turn use different shift values. However the ciphertext produced by the classical ciphers reveal significant statistical information about the plaintext and hence can be broken easily. To reduce the statistical dependence between the plaintext and the ciphertext modern ciphers employ variety of techniques in combination with substitution and transposition.

Though the initial usage of cryptography was limited to providing confidentiality only, this has changed with the invention of computers and other communication devices. In general modern cryptography mainly addresses the following problems:

- **Confidentiality.** Only the intended recipient should be able to recover the plaintext from the ciphertext.
- **Integrity.** The message recipient should be able to detect if the original message has been altered during the transmission.
- **Authentication.** The message recipient should be able to verify from the message the identity of the sender.
- **Non-repudiation.** The sender should not be able to deny sending of the message later.

In modern cryptography only the cryptographic *key* (a parameter to the algorithm which determines the output) is kept secret. The algorithm itself is assumed to be available to the attacker. The cryptographic algorithms are mainly classified into three types: symmetric key cryptography, public key cryptography and hash functions.

- **Symmetric Key Cryptography.** In symmetric key cryptography, the sender and the receiver use the same key. For the security of these algorithms, both the parties need to agree upon a key before the communication can start. The symmetric key algorithms are generally divided into two categories:
  - **Block ciphers.** These ciphers divide the plaintext into several blocks (based on the block length of the cipher) and encrypt each block independently using the secret key. Examples of block ciphers include DES (Data Encryption Standard) [Sta77], Advanced Encryption Standard (AES) [FIP01] etc.

  - **Stream ciphers.** These ciphers encrypt the plaintext one bit at a time using the key stream generated by some form of feedback mechanism. Examples of stream ciphers include RC4, Salsa20 etc.

- **Public Key Cryptography.** The public (or asymmetric) key cryptography involves two keys: public key and private key. It depends on existence of one-way trapdoor functions, which are easy to compute but computing their inverse function is computationally difficult. For example, given two integers, finding their product is an easy problem. However, given a number, it is difficult to find the factors of it (so called factorization problem). The public key algorithms are mainly divided into two categories:

  - **Encryption algorithms.** The asymmetric encryption algorithms solve the problem of confidentiality by employing two set of keys. If Alice wants to send a message to Bob, she uses Bob's public key to encrypt the message. Upon receiving the message, Bob uses his private key to decrypt the ciphertext. Examples of public key encryption algorithms include RSA [RSA78], Elgamal etc.
  - **Digital signature algorithms.** The digital signature algorithms solve the problem of authentication. Alice uses her private key to sign a message and sends it to Bob. Bob can verify the signature using Alice's public key. Examples of digital signature algorithms include DSA, Elgamal etc.

- **Hash Functions.** The hash functions are computationally efficient mappings which take arbitrary length strings as input and output a string of fixed length called hash value. For a cryptographic hash function it is required that given a hash value it is computationally infeasible to compute the message which hashes to it. Hash functions are widely used in digital signatures and to provide data integrity. Examples of hash functions include SHA-1, SHA-2, SHA-3, MD5 etc.

**Cryptanalysis.** The goal of the cryptanalyst (who does cryptanalysis) is to break a cryptographic algorithm. For example, in case of symmetric key algorithm, he could try to find the secret key and hence recover all the past and future communications using that key. This process is generally referred to as an *attack*. Normally, an attacker can always perform a search on all possible key candidates and can recover the correct key (called *brute-force attack*). Hence, the most important requirement in designing a secure cipher is to make it infeasible for an attacker to perform brute-force search. However what constitutes an attack is much broader than just recovering the secret key. Any weakness found in the algorithm that reduces the complexity of an attacker compared to brute-force is also referred to as an attack. Assuming that the knowledge of the algorithm is public, the cryptanalytic attacks are divided into four types:

- **Ciphertext-only attack.** The attacker has access to several ciphertexts

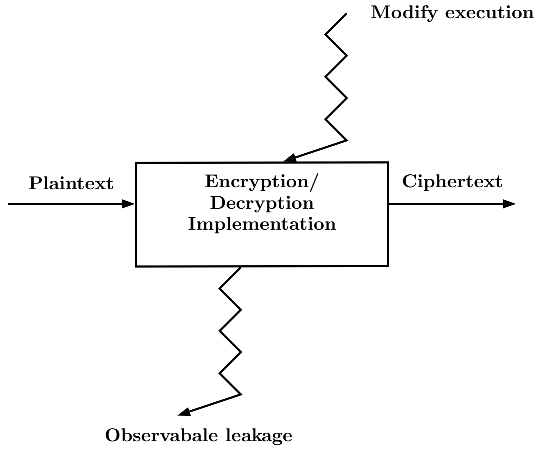


Figure 1.2: The outline of implementation attacks

produced using the same secret key. The goal of the attacker is to find the plaintexts corresponding to them or even the secret key.

- **Known-plaintext attack.** The attacker has access to several plaintexts and their corresponding ciphertexts. His job is to find the secret key using those pairs.

- **Chosen-plaintext attack.** The attacker can choose plaintexts of his own choice and can get the corresponding ciphertexts. Note that this attack is more powerful than known-plaintext attack.

- **Adaptive-chosen-plaintext attack.** This can be considered as a special case of chosen-plaintext attack. Here the attacker can chose the plaintexts based on the results from previous chosen-plaintexts.

## 1.2 Implementation Attacks and Countermeasures

Implementation attacks as the name suggests exploit the physical implementations of cryptographic algorithms on electronic devices. Therefore, this kind of attacks are methodically very different from "traditional" cryptanalysis, which essentially focuses on finding secret keys in a black box model given only pairs of plaintexts and ciphertexts by exploiting the mathematical weaknesses in the algorithms.

The implementation attacks are categorized based on two criteria. The first criteria is whether the attacks are passive which work based on observable leakage or active which modify the execution environment (Refer to Figure 1.2 ).

- **Passive attacks.** In passive attacks the attacker tries to recover the secret key using the observable phenomena from a cryptographic implementation. This include timing attacks, i.e. attacks exploiting measurable differences in the execution time of a cryptographic algorithm or a specific operation it is based upon [Koc96, Hey98]. A more sophisticated class of attacks is power analysis attacks, which aim to deduce information about the secret key from the power consumption of the device while a certain operation is carried out [MOP07]. A third class are electromagnetic (EM) attacks, which exploit the relationship between secret data and EM emanations produced by the device [AARR03].

- **Active attacks.** In active attacks the attacker modifies the execution environment of the cryptographic device. For example, he can manipulate the inputs so as to make the device behave in an abnormal way. This abnormal behavior can then be used to recover the secret key. Examples of active attacks include fault attacks [BDL97, BS97], which induce faulty inputs into the device.

The second criteria used to categorize the implementation attacks is the interface of the device exploited by the attacker. There are three different attacks based on these criteria:

- **Invasive attacks.** In an invasive attack the attacker has the full control over the cryptographic device. He can practically depackage the chip and can access/control any part of the chip using a probe. The invasive attacks can be passive or active based on whether the probe is used to only observe the behavior of the chip or alter the functionality by changing the signals. Though these attacks are strongest form of implementation attacks, they are however very costly to perform as they require very expensive setup.

- **Semi-invasive attacks.** In a semi-invasive attack the attacker can also depackage the chip. However, unlike the invasive attacks the attacker has limited control over the device and the passivation layer of the chip remains intact. For example, he can't make a direct contact to the chip surface or use probing. A passive semi-invasive attack involves reading the content of the memory, whereas an active semi-invasive attack induces faults in the device. Though these attacks are relatively cheaper than the invasive attacks, they still require significant effort as finding the exact location on the chip to attack requires time and money. For a comprehensive treatment of invasive and semi-invasive attacks please refer to the PhD thesis of Sergei P. Skorobogatov [Sko05].

- **Non-invasive attacks.** In a non-invasive attack the attacker can only observe/control the accessible interfaces of the device. These are relatively inexpensive to perform requiring as low as a $1000 setup. In particular, passive non-invasive attacks also called *side-channel attacks* are very easy to mount and hence gained significant attention from the research community. Side-channel attacks include timing attacks, power analysis attacks, EM attacks *etc.* On the other hand, active non-invasive attacks induce faults into the device without unpacking it (for *e.g.* by inducing clock glitches). This thesis solely focuses on side-channel attacks and their countermeasures.

### 1.2.1 Side-channel Attacks

Though side-channel attacks were believed to be known for some time, they were first documented by Paul Kocher in 1996 [Koc96]. Most prominent side-channel attacks include timing attacks [Koc96], power analysis attacks [KJJR11] and electromagnetic emission attacks [AARR03].

- **Timing attacks.** Cryptographic algorithms often take different amounts of time for different inputs. For example, an exponentiation operation is normally implemented using square-and-multiply method. In this method, the exponent is represented as a series of binary digits. If the exponent bit is 0 we only perform square operation, whereas if the bit is 1 we perform both the square and multiply operations. As the timing requirement for both the bits is different, the attacker can differentiate between the two and hence recover the secret key if used in an exponentiation. Similar attacks are also possible which exploit the time it takes for cache hit/miss (called cache-timing attacks).

- **Power analysis attacks.** In general the power consumption of a device depends on the processed data and the performed operations. For example, if a data bit changes from 0 to 1 or 1 to 0 as a result of an operation, it dissipates more power than when it does not change. This variation in the power consumption can be exploited by the attacker to recover the secret key. For a most comprehensive treatment of power analysis attacks, refer to [MOP07].

- **EM attacks.** Similar to the power consumption, the electromagnetic radiation of the device is also dependent on the processed data. Hence by observing the EM radiation emitting from the device while the secret key is being processed, the attacker can recover the secret key.

Though timing attacks are the easiest side-channel attacks to mount, they are also easy to circumvent. Due to their simplicity and effectiveness, the power analysis attacks received significant attention from the research community. Throughout this thesis the words side-channel attacks and power analysis attacks are used interchangeably as most of the techniques used in power analysis attacks can also be used to mount EM attacks as well.

**Simple power analysis (SPA) attacks.** The SPA attacks try to recover the secret key by visually inspecting the power measurements from the cryptographic device. For example, if we consider the square-and-multiply method of implementing the exponentiation, the pattern in the power trace corresponding to the square operation (exponent bit 0) will be different from the pattern corresponding to square

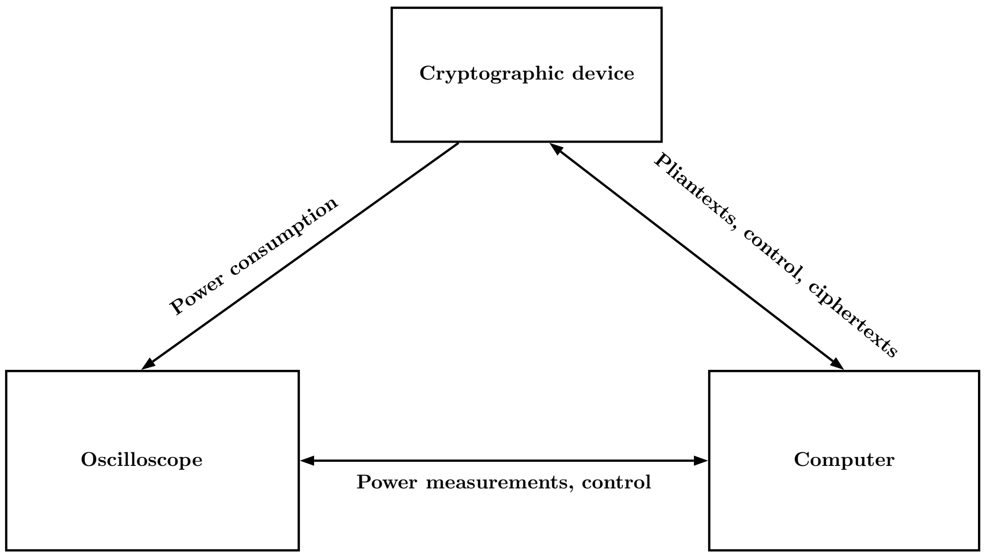


Figure 1.3: The outline of a DPA attack

and multiply operations (exponent bit 1). By observing one or few such traces from the measurement, the attacker could recover the full secret key. However, these attacks can be usually quite challenging in practice (due to noise) and require full knowledge of the cryptographic implementations.

**Differential power analysis (DPA) attacks.** The DPA attacks in contrast to the SPA attacks require little knowledge about the cryptographic implementations and work based on divide-and-conquer approach. In cryptographic algorithms, the operations are normally performed on a part of the secret key. Therefore, the attacker can compute predictions over all possible values of a key chunk. Then he can recover the secret key using the statistical correlation between the observed power values and the estimated values for all the possible key guesses. Examples of the statistical techniques used in DPA attacks include distance-of-means, correlation coefficient etc. These attacks can be successful even with extremely noisy power measurements and hence are a big threat to the security of the cryptographic devices.

A typical DPA attack works as shown in Figure 1.3. The cryptographic device implements a certain cryptographic algorithm (*e.g.* AES). To measure the power consumption of the device during its operation, a small resistor (1 $\Omega$) is connected in series to the power supply of the device. The device is then connected to an oscilloscope which records the power consumption values and sends them to the computer. The computer sends the plaintexts to the device and receives the corresponding ciphertexts after the encryption. Based on the ciphertexts and their corresponding power measurements, we then perform the DPA attack.

Generally side-channel attacks follow a two-step procedure: Finding the appropriate leakage model for the device in question and recovering the secret key using appropriate distinguisher. Depending on the choice of the leakage model the side-channel attacks are divided into two categories:

- **Profiled attacks.** The profiled attacks work in two phases: off-line phase and on-line phase. In the off-line phase the attacker builds the device leakage model for the device in question. There exist several techniques in the literature to build such models: Gaussian templates [CRR02], linear regression [SLP05] etc. In the on-line phase the attacker has access to one or more power traces corresponding to the same key. By using the leakage model built in the off-line phase and appropriate statistical distinguisher, the attacker differentiates between the correct key and the wrong key candidates.

- **Non-profiled attacks.** In case of non-profiled attacks the attacker chooses the leakage model based on a-priori information about the device and hence there is no off-line phase. Examples of such leakage models include Hamming weight model, Hamming distance model etc. In Hamming weight model the device leaks the Hamming weight (i.e. the number of 1's in the binary representation) of the data under processing. On the other hand in Hamming distance model the leakage is proportional to the hamming distance between the old and new data (i.e. the number of positions at which the old and new data differs). The rest of the attack works similar to the profiled attacks.

There exist several distinguishers useful in a side-channel key recovery attack. We recall two most widely used distinguishers below:

- **Distance-of-means.** For each possible key candidate, the attacker divides the power traces into two parts based on whether a particular bit of an intermediate variable is 1 or 0. Then he computes the difference of the means of both the parts. The key candidate with the highest difference would be the correct key [KJJ99].

- **Pearson's correlation coefficient.** For each possible key candidate, the attacker computes the predicted power consumption based on some leakage model. Then each of these predicted values are correlated with the observed power values from the actual measurements. The key candidate with the highest absolute correlation would be the correct key [BCO04].

Both the distinguishers recalled above can only reveal linear dependencies between the predicted values and the power consumption values. Recently there were efforts to find a generic side-channel distinguisher which can reveal all kind of dependencies between the predicted and actual values. One such distinguisher is Mutual Information Analysis (MIA), which works based on the information theoretic metric Mutual Information [GBTP08, VS09].

### 1.2.2 Countermeasures against Side-channel Attacks

Ever since the introduction of Side-Channel Analysis (SCA) attacks in the late '90s, there has been a massive body of research on finding effective countermeasures to thwart these attacks, in particular the highly effective Differential Power Analysis (DPA) attacks. From a high-level perspective, DPA countermeasures aim to either randomize the power consumption (which can be done in both the time and amplitude domain) or make it completely independent from the processed data. The goal of both approaches is to eliminate (or, at least, reduce) the correlation between the power consumption and the key-dependent intermediate variables processed during the execution of a cryptographic algorithm. The most widely used countermeasures against side-channel attacks fall into two categories: hiding and masking. These countermeasures in turn can be applied at different stages. Algorithmic countermeasures (also called software countermeasures) modify the cryptographic algorithm in such a way that the leakage from the sensitive variables is reduced. On the other hand hardware countermeasures employ several techniques to reduce the data dependent power leakage from the cryptographic device. There also exist countermeasures at the protocol level which essentially limit the number of times a particular key is used in a cryptographic operation, thus restricting the number of measurements available to the attacker in a DPA attack. In practice, the implementations of cryptographic systems use a mixture of several countermeasures depending on the required security level [CCD00]. This thesis mainly focuses on the algorithmic countermeasures.

**Hiding.** There are two ways a countermeasure can eliminate the dependency between the power consumption of the device and the processed data:

- Randomize the power consumption in each clock cycle.
- Ensure that the device power consumption is uniform in each clock cycle.

There exists several ways to achieve these, some of which are recalled below.

- **Random delays.** This technique involves inserting random delays in the middle of the execution of a cryptographic algorithm. This ensures that the attacker cannot align the traces, which is required for a successful DPA attack.
- **Dummy operations.** Here, unrelated dummy operations are inserted inside the cryptographic algorithm. Similar to random delays, this technique also induces misalignment in the traces thus increasing the required effort for the attacker.
- **Shuffling.** This technique involves re-ordering the sequence of operations in such a way that the final output of the algorithm does not change. For example, the order of execution of S-boxes in AES can be randomized without affecting the final result. This randomization is mainly performed using two approaches: Using a random permutation (called RP method) or using a random start index (RSI). Based on the level of randomization, the shuffling can amplify the noise in the device and hence increase the effort required by the attacker. For a comprehensive treatment of shuffling countermeasure refer to [VMKS12] and [RPD09].

**Masking.** The advantage with hiding countermeasures is that they are relatively efficient to implement in practice. However, most of these techniques (except for some variants of shuffling) are ad-hoc and do not provide any concrete security guarantees. To overcome this problem we use masking, which can be proven under certain assumptions on the device leakage model. In order to circumvent DPA attacks using masking, we divide the secret value into two shares: a mask generated randomly and the masked value of the secret. However, this approach can still be attacked via a second-order DPA attack involving two operations corresponding to the two shares of the secret [JPS05]. To circumvent this, we use two randomly generated masks with a total of three shares including the masked value of the secret. In general, a $d$-th order masking scheme is vulnerable to a $(d+1)$-th order DPA attack involving all $d+1$ shares of the secret. These attacks are called Higher-order DPA attacks (HODPA) and the corresponding masking is called Higher-order masking.

Masking, depending on the involved operations, can be either Boolean, arithmetic, or multiplicative. When used to protect a cryptographic algorithm that performs a mixture of these operations, it is necessary to convert the masks from one form to the other in order to be able to produce the correct result at the end of the algorithm.

## 1.3 Contributions

This thesis focuses on securing the cryptographic implementations against side-channel attacks. In particular, it examines the masking countermeasure and proposes new algorithms to protect implementations of symmetric key cryptosystems. It also analyzes previously published masking schemes and provides new insights to improve their efficiency and security.

We start with a review of some of the prominent masking schemes in Chapter 2. We first provide a general introduction to masking and the problem of mask conversion. We then examine Boolean masking in detail and review the state of the art. Finally we recall the existing solutions to convert between Boolean and arithmetic masking.

In chapter 3 we give our first solution to the problem of conversion between Boolean and arithmetic masking secure against any order. To set the context, we first show that existing first-order solutions can not directly be extended to protect against higher-order attacks. We then give two solutions to perform addition on Boolean shares with varying complexity. Using these solutions we develop the conversion algorithms from arithmetic to Boolean masking as well as Boolean to arithmetic masking. We prove the security of all the proposed algorithms in a well-known security model. We also give the implementation results of all the proposed algorithms when applied to HMAC-SHA-1 on a 32-bit microcontroller. This is a joint work with Jean-Sébastien Coron and Johann Großschädl and published in the proceedings of CHES 2014 [CGV14].

All the existing solutions to convert from arithmetic to Boolean masking have linear complexity in the size of the masked operand. In chapter 4 we give new algorithms with logarithmic complexity thus gaining exponential improvement. We also show that our new techniques can be naturally extended to higher-order masking and hence require lesser time compared to the solutions given in chapter 3. We give the experimental validation of our improved algorithms with two real-world examples: HMAC-SHA-1 and light weight block cipher SPECK. This is a joint work with Jean-Sébastien Coron, Johann Großschädl and Mehdi Tibouchi and will appear in the proceedings of FSE 2015 [CGVT15].

In chapter 5 we further improve the running time of the first and second-order secure conversion algorithms by employing lookup tables. We first give two straightforward algorithms using the generic second-order secure countermeasure. However, these algorithms quickly become inefficient as they require a lookup table of $2^n$ entries for the masked operands of $n$-bit. To overcome this challenge we propose to use divide-and-conquer approach, which reduces the size of the lookup tables to a manageable size. This is a joint work with Johann Großschädl and published in the proceedings of SPACE 2013 [VG13] and COSADE 2015 [VG15].

In chapter 6 we study the fast and provably secure higher-masking of AES S-box proposed by Kim *et al.* at CHES 2011 [KHL11]. Their scheme uses composite field methods to accelerate the inversion operation in $\mathbb{F}_{2^8}$. However, as we show their $n$-th order secure scheme is actually insecure against attacks of order $n/2+1$. We then propose a method to fix this flaw. This is a joint work with Junwei Wang, Johann Großschädl and Qiuliang Xu and published in the proceedings of CT-RSA 2015 [WVGX15].

In chapter 7 we examine the validity of security proofs in different leakage models. We show that a security proof given in one model is no more secure if the device leaks in a different model. This result emphasizes the need to re-evaluate the masking schemes when porting an implementation from device leaking in one model to another. This is a joint work with Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner and Matthieu Rivain and published in the proceedings of COSADE, 2012 [CGP+12b].

# Chapter 2

# Masking

Masking is, besides hiding, the most widely used countermeasure to thwart Differential Power Analysis (DPA) attacks on symmetric cryptosystems. The main advantage with the masking schemes is that one can formally prove their security under certain assumptions on the device leakage model. In this chapter we recall several prominent masking schemes published in the literature along with their security guarantees and limitations.

**Contents**

## 2.1 Introduction

Masking aims to conceal each sensitive intermediate variable $x$ with a random value $x_2$, called mask [CJRR99]. This means that the sensitive variable $x$ is represented

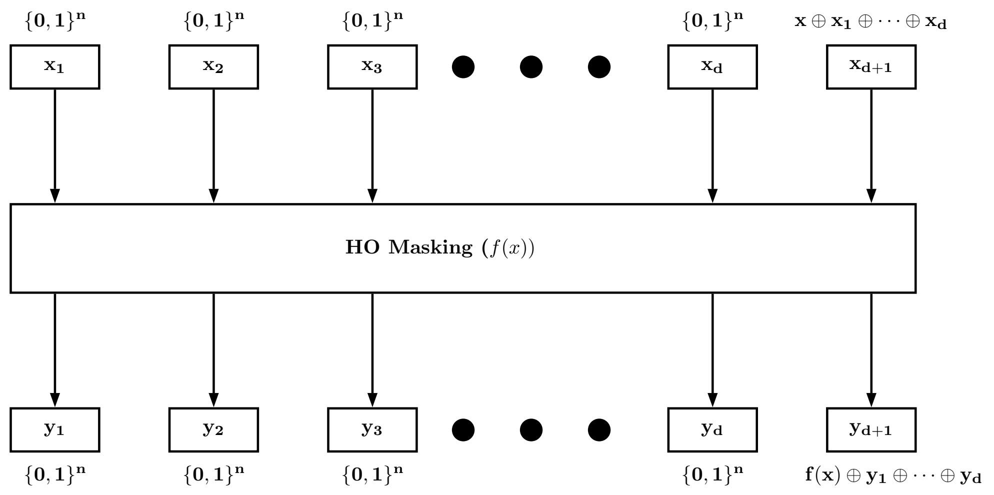


Figure 2.1: Evaluating $f(x)$ secure against $d$-th order attacks

by two shares, namely the masked variable $x_1 = x \oplus x_2$ and the mask $x_2$. The shares need to be manipulated separately throughout the execution of the algorithm to ensure that the instantaneous power consumption of the device does not leak any information about $x$. Indeed, a straightforward DPA attack may yield $x_1$ or $x_2$ (both of which appear as random numbers to the attacker), but knowledge of $x_1$ alone or $x_2$ alone does not reveal any information about the sensitive variable $x$.

One of the main challenges when applying masking to a block cipher is to implement the round functions in a way that the shares can be processed independently from each other, while it still must be possible to recombine them at the end of the execution to get the correct result. This is fairly easy for all linear operations, but may introduce significant overheads for the non-linear parts of a cipher, i.e. the S-boxes. In addition, all round functions need to be executed twice (namely for $x_1$ and $x_2$, where $x = x_1 \oplus x_2$), which entails a further performance penalty. Another problem is that a basic masking scheme as described above is vulnerable to a so-called second-order DPA attack, in which an attacker combines information from two leakage points. Namely, he exploits the side-channel leakage originating from $x_1$ and $x_2$ simultaneously [OMHT06]. Such a second-order DPA attack can, in turn, be thwarted by second-order masking in which each sensitive variable is concealed with two random masks and represented by three shares. In general, a $d$-th order masking scheme uses $d$ random masks to split a sensitive intermediate variable into $d+1$ shares $x_1, x_2, \ldots, x_{d+1}$ satisfying $x_1 \oplus x_2 \oplus \cdots \oplus x_{d+1} = x$, which are processed independently. In this way, it is guaranteed that the joint distribution of any subset of up to $d$ shares is independent of the secret key. Only a combination of all $d+1$ shares (i.e. the masked variable $x_1 = x \oplus x_2 \oplus \cdots \oplus x_{d+1}$ and the $d$ masks $x_2, \ldots, x_{d+1}$) is jointly dependent on the sensitive variable. However, given a sufficient amount of noise, the effort for attacking a higher-order masked implementation increases exponentially with $d$ [CJRR99].

To evaluate a function $f$ on the sensitive variable $x$ against attacks of order $d$, we use the method shown in Figure 2.1. We first generate $d$ input shares $(x_1, \cdots, x_d)$ and $d$ input shares randomly $(y_1, \cdots, y_d)$. The $d+1$-th input share is computed as: $x_{d+1} = x \oplus x_1 \oplus \cdots \oplus x_d$. Then, the higher order masking produces $d+1$-th output share $y_{d+1}$ such that $y_{d+1} = f(x) \oplus y_1 \oplus \cdots \oplus y_d$.

#### Provably Secure Masking.

As noted previously, the advantage with masking schemes is that one can prove the security of the corresponding algorithms based on certain realistic assumptions on the device leakage model. The prominent models for proving the masking schemes are: Hamming weight model, Hamming distance model and probing model. Proving security against first-order attacks is easy since one can list all the intermediate variables occur in the algorithm and then show that none of them are dependent on the secret key (i.e., their distribution is independent of the secret key). This approach can be generalized to any order masking. For example in case of second order masking, we can show that no pair of intermediate variables depends on the secret key. However, as the order increases, the number of tuples that need to be considered grows exponentially. To counter this, we use a different approach proposed by Ishai, Sahai and Wagner in [ISW03]. The main principle behind ISW approach is that if the distribution of any set of $t$ intermediate variables can be simulated without the knowledge of the original inputs, then the scheme is secure against attacks of order $t$. To achieve this, one can iteratively generate a subset of the input shares that are sufficient to simulate the distribution of $t$ intermediate variables. If the number of input shares required is less than the actual number of shares, then we can generate those shares randomly.

#### Mask Conversion.

Depending on the operation to be protected, a masking scheme can either be Boolean (using logical XOR), arithmetic (using modular addition/subtraction) or multiplicative (using modular multiplication). To successfully “unmask” the variable at the end of the algorithm, one has to track the change of the masked secret value during the execution of the algorithm. If an algorithm contains two of the three afore-mentioned operations (i.e. XOR, modular addition/subtraction, modular multiplication), the masks have to be converted from one form to the other, keeping this conversion free from any leakage. Goubin introduced secure methods to convert between first-order Boolean and arithmetic masks in [Gou01]. Coron and Tchulkine improved the method for switching from arithmetic to Boolean masking in [CT03], which was recently further improved by Debraize in [Deb12]. There also exist solutions for converting between arithmetic and multiplicative masking of higher-order [GPQ10, GPQ11a, GPQ11b].

**Boolean vs Arithmetic Masking.**

Boolean masking is widely-used countermeasure for cryptographic algorithms that use only linear operations over the field $\mathbb{F}_2$ and non-linear S-boxes (e.g. DES and AES). However, if an algorithm includes arithmetic operations (such as IDEA [LM90], RC6 [CRRY04], and SHA-1 [NIS95]), a masking scheme that is compatible with the arithmetic operation must be used [CJRR99]. For example, if $x_3 = x_1 + x_2$ must be computed securely, we can mask both $x_1$ and $x_2$ arithmetically by writing $x_1 = A_1 + r_1$ and $x_2 = A_2 + r_2$ for some random values $r_1$ and $r_2$. Then, instead of computing the sum $x_3$ directly, we can add the two shares separately, which results again in two arithmetic shares for $x_3 = (A_1 + A_2) + (r_1 + r_2)$.

Besides IDEA, RC6 and SHA-1, there also exist many other algorithms that execute both arithmetic (e.g. modular addition) and logical operations. Examples include ARX-based block ciphers like XTEA and Threefish, the SHA-3 finalists Blake and Skein, as well as all four stream ciphers from the e-Stream software portfolio. Hence, techniques to protect both kinds of operation are of practical importance. There exist two approaches to solve this problem.
**Using mask conversion.** This is a three step process as given below:

1. Convert Boolean shares to corresponding arithmetic shares.
2. Perform addition on arithmetic shares.
3. Convert the result back to Boolean shares.

Note that this approach requires that the mask conversion itself is also secure against first-order (resp. higher-order) attacks.
**Addition on Boolean shares.** In this approach, use only one Boolean masking and employ secure algorithms to perform the addition directly on the shares.

While there exist some papers about the first method, the second approach has, surprisingly, not been studied in detail. The decision whether to apply the conversion or not depends on the target cryptographic algorithm.

**Organization.**

The rest of the chapter is organized as follows. We first describe the security guarantees provided by masked implementations in Section 2.2. Then we review the existing techniques to evaluate cryptographic implementations in Section 2.3. We then recall some of the prominent results related to Boolean masking (used to protect AES, DES etc.) in Section 2.4. Goubin's algorithms to convert between Boolean and arithmetic masking are given in Section 2.5. The improved solutions to convert from arithmetic to Boolean masking using lookup tables are recalled in Section 2.6.

## 2.2 Masking Security Guarantees

The main principle behind masking schemes is to randomize the secret data manipulated by the leaking device. This is achieved by splitting each sensitive variable into $d$ shares (where $d-1$ shares are generated randomly) and performing all the operations on the shares independently. For a masking scheme to be secure, the distribution of any combination of $d-1$ shares should be independent of the sensitive variable. To recover the secret key, the attacker then needs to combine the leakages from at least $d$ shares, which requires estimating $d$-th order moment of the leakage distribution. Given sufficient noise and provided the leakages from different shares are independent, computing such a distribution becomes exponentially hard in the number of shares. In this section, we give a brief overview of the security guarantees provided by masked implementations and their limitations.

The first formal study of masking was conducted by Chari *et al.* in [CJRR99]. They show that in the presence of noisy leakage, the complexity of a single bit Differential Power Analysis attack increases exponentially with the number of shares. To be more precise, let every bit $b$ computed in the algorithm is split into $d$ shares as follows: generate $d-1$ bits randomly (let them be $r_1, r_2, \cdots, r_{d-1}$) and compute the $d$-th share as $r_1 \oplus r_2 \oplus \cdots r_{d-1} \oplus b$. Also assume that each of these shares are part of a different word and other bits of the word are chosen uniformly at random. By making reasonable assumptions on the device leakage model, they show that the effort required by the attacker increases exponentially in $d$.

**Power model.** The power consumption of CMOS devices mostly depends on the changes in the logic circuits, for *e.g.* change in the values of the registers, RAM, address bus, data buts etc. In a chip, each clock edge triggers a series of operations which contribute to the overall power consumption. If we ignore the effect due to glitches, the instantaneous power consumption can be approximately modeled as the sum of the power consumption due to all the events occurred during that clock cycle. Hence, for any share we have

$$P = P_b + P_C + R$$

where $P$ is the instantaneous power consumption of the device, $P_b$ is the power consumption corresponding to the particular bit $b$, $P_C$ is the distribution of the power contributions where bit $b$ and other bits of the share are involved and $R$ is the distribution of the noise. For simple operations, $P_C$ is negligible and can be ignored. The noise $R$ can be modeled using normal distribution with mean $\mu$ and variance $\sigma^2$.

Let us assume that the adversary can record several power consumption measurements for random inputs. Every measurement sample contains information about the $d$ shares at different instances in time. Let the distribution of the power consumption for each of these shares be $Z_1, Z_2, \cdots, Z_d$. Also assume that $Z_i = A_i + X_i$, where $A_i$ is the contribution from the actual bit $r_i$ and $X_i$ is the additive factor with a distribution $R$, which also contains the noise. The value of $A_i$ can be either 0 or 1 with equal probability (i.e. 1/2). Hence, the power consumption profile for $A_i$ is different depending on the value of $r_i$. Let these distributions be $D_1$ (when $b=0$) and $D_2$ (when $b=1$).

To successfully recover the secret bit $b$, the adversary needs to distinguish between the two distributions $D_1$ and $D_2$. The result from [CJRR99] shows that the attacker needs at least $n^{k/2-4\delta}$ samples to distinguish between $D_1$ and $D_2$ with a probability $n^{-\delta}$, where $n = \sigma^2$ is the variance of the Gaussian noise. The main theorem from [CJRR99] is recalled below.

**Theorem 2.1.** *Let $\delta$ be a constant. Given distributions $D_1$ and $D_2$ defined above, any adversary which has access to $m < n^{k/2-4\delta}$ samples ($n = \sigma^2$) from these two distributions, has probability at most $n^{-\delta}$ of distinguishing $D_1$ and $D_2$.*

Though the analysis performed by Chari *et al.* improved the confidence in the security of carefully implemented masking schemes, it had an important limitation. Their analysis is only valid for individual operations and is not applicable for the full block cipher.

**Probing model.** In [ISW03] Ishai, Sahai and Wagner (ISW) initiated a theoretical study of securing circuits against an adversary who can probe a limited number of wires in the circuit. In probing model, the adversary is allowed to access at most $t$ wires in the circuit without learning anything about the secret key. They proposed a method to transform any circuit with $n$ gates into an equivalent circuit of $\mathcal{O}(n^2 t)$ gates that is secure against probing attacks. However, the probing model has an important limitation that it does not consider an attacker who can exploit the leakage from the complete implementation.

In [FRR$^+$10], Faust *et al.* proved the security of ISW method in two more general leakage models. In the first model, the leakage function is considered to be computationally bounded (belonging to $\mathsf{AC}^0$ complexity class) and requires a leak-free hardware component. However, Rothblum [Rot12] in his work gave a method to compute under $\mathsf{AC}^0$ leakage without the additional requirement of a leak-free hardware component. In the second model, it is assumed that the implementation leaks the bits of the circuit state perturbed by independent binomial noise. To be more precise, each bit is flipped with probability $p$ and remains unchanged with probability $1-p$. Though these results prove the security of masking in the presence of global leakage, the considered models seem impractical.

**Security of masking in practice.** In [SVO$^+$10] Standaert *et al.* analyzed the practical security of masking implementations. With the help of information theoretic framework introduced by Standaert *et al.* in [SMY09], they show that exponential security of masking schemes is only possible when there is sufficient noise. Namely, given a $d$-th order masked implementation, the number of traces required for an attacker to recover the secret key is proportional to $(\sigma^2)^{d/2}$, where $\sigma^2$ is the variance of the noise present in the device. Though this analysis considered only Hamming weight leakage model for an adversary who can perform a worst-case template attack, it confirms the theoretical claims about exponential security provided by masking schemes in practice.

**Security of masking against global leakage.** A first attempt to prove the security of a masked implementation of the full block cipher under realistic leakage models was carried out by Prouff and Rivain in [PR]. They consider only computation leaks model, which assumes that each computation carried out by the device reveals a leakage function of the data that is actually used during the current computation. They assume that these leakage functions are noisy and hence every elementary operation leaks only a noisy function of it's input, where the noise can be changed depending on the required security level. Furthermore, they assume the existence of a leak-free component to refresh the masks. Based on these assumptions, they prove that for a $d$-th order masking, the information about the sensitive data provided by the leakage from the full block cipher implementation can be made negligible in the masking order.

Later Duc *et al.* gave a reduction proof from the noisy leakage model by Prouf and Rivain to the ISW probing model [DDF14]. As a result, the requirement of leak-free component from Prouff-Rivain has been removed. This result has an important implication. All the masking schemes (including the ones given in this thesis) proven in the ISW probing model indeed provide exponential security in the number of shares against side-channel attacks provided the measurements have sufficient noise and the leakages from different shares have independent distributions.

**A note on the independence assumption.** In practice, it is not always possible for an implementation to ensure independence between the leakages from different shares. For example, in case of software implementations, if the device leaks in the Hamming distance model, the independence condition may not be satisfied, which could reduce the effort required by the attacker (as we show in Chapter 7). Namely, for a device leaking in the Hamming distance model, one might need twice the number of shares compared to the device leaking in the Hamming weight model for achieving same level of security [BGG$^+$14]. On the other hand, glitches that occur in the circuits of masked gates can invalidate the independence assumption in hardware implementations [MPG05, BNN$^+$12, MPL$^+$11]. In [DFS15], Duc *et al.* provided tools to analyze the impact of such non-independent leakages on the security of the masked implementations.

## 2.3 Evaluating Security of an Implementation

In this section, we summarize the techniques to evaluate the security of an implementation against side-channel attacks. We first describe the powerful Differential Power Analysis (DPA) attack and Correlation Power Analysis (CPA) attack in the context of first-order security. We then review template attacks which require additional methods to profile the device leakage model. Thereafter we recall the methods to defeat masked implementations using higher-order DPA attacks. Then we describe Mutual Information Analysis (MIA) attacks, which can be applied to both unmasked and masked implementations. Finally, we give an overview of the leakage detection tests.

### 2.3.1 DPA Attacks

DPA attacks are extremely powerful since they do not require full knowledge of the implementation and can reveal the secret key even when the power measurements have significant noise. Generally a DPA attack consists of five steps as given below.

1. **Choosing the intermediate variable.** The first step in a DPA attack is to decide on an intermediate variable that needs to be examined. This is typically a function of the plaintext (or ciphertext) and a part of the secret key. For example, in AES one can choose the output of an S-box i.e. $S(p_i \oplus k_i)$ where $p_i$ and $k_i$ are the i-th plaintext and key bytes. A general rule of thumb is to pick a variable that is the result of a non-linear operation.

2. **Collecting the power traces.** In the second step, the attacker needs to collect the power traces for different values of the selected intermediate variable while the device is performing the required cryptographic operation. Each power trace $PT$ is a vector of power consumption values over different instances in time. Hence, this step produces a matrix $T$ of size $N \times PT$, where $N$ is the number of power traces acquired.

3. **Computing hypothetical intermediate values.** In the next step, the attacker computes the value of the intermediate variable for each key candidate. For example, in case of AES if the S-box output is the targeted intermediate variable, the value of $v_{i,j} = S(p_i \oplus j)$ is computed for each possible value of $k$, *i.e.* $0 \leq j \leq 255$ and for the all the plaintexts $p_i : 0 \leq i \leq N$ [1]. As a result we have a matrix $V$ of size $N \times K$, where $K$ is the number of all possible key candidates.

4. **Estimating hypothetical power consumption values.** In this step, the hypothetical intermediate values computed in Step 3 are mapped to the hypothetical power consumption values. This requires the knowledge of the power model corresponding to the device. The widely used models are: Hamming weight model and Hamming distance model. This step produces a matrix $H$ of size $N \times K$.

5. **Comparing the hypothetical power values with the power traces.** In the final step, the hypothetical power values computed in step 4 are compared with the actual power traces collected in step 2. Namely, each column $H_i$ in $H$ is compared with each column $T_i$ in $T$ to produce a matrix $R$ of size $K \times PT$, where the element $R_{i,j}$ corresponds to the comparison between columns $H_i$ and $T_j$. Then the highest value $R_{ck,ct}$ in the matrix $R$ corresponds to the correct key (for the column $H_{ck}$). Based on the used comparison function, there exists different types of DPA attacks, some of which are recalled below.

**Single-bit DPA attack.** In a single-bit DPA attack the hypothetical power consumption values in step 4 are computed based on the Hamming weight or Hamming

[1]For simplicity, we consider only one byte of the plaintext and the secret key.

distance of a single bit of the targeted intermediate variable. For bit $l$ of the intermediate variable $v_{i,j}$ we have:

$$H_{i,j} = f(v_{i,j,l}) \text{ where } f \text{ is replaced by } HW \text{ or } HD$$

Then we divide each row $T_i$ in $T$ into two groups $S_{0,j}$ and $S_{1,j}$ for each possible value of key $(j)$ as follows:

$$\begin{aligned} S_{0,j} &= \{T_i | H_{i,j} = 0\} \\ S_{1,j} &= \{T_i | H_{i,j} = 1\} \end{aligned}$$

Now we compute the average power trace for each of the sets:

$$\begin{aligned} A_{0,j} &= \frac{1}{|S_{0,j}|} \sum_{S_{0,j,l} \in S_{0,j}} S_{0,j,l} \\ A_{1,j} &= \frac{1}{|S_{1,j}|} \sum_{S_{1,j,l} \in S_{1,j}} S_{1,j,l} \end{aligned}$$

Then we compute the difference between $A_{0,j}$ and $A_{1,j}$ which is stored in $R_j$.

$$R_j = |A_{1,j} - A_{0,j}|$$

The value $j$ corresponding to the maximum value in the matrix $R$ would be the correct key.

**Multi-bit DPA attack.** In a multi-bit DPA attack the hypothetical power consumption values are computed based on the Hamming weight or Hamming distance of multiple bits of the targeted intermediate variable. The comparison function is also changed as a result. There exits two variants of multi-bit DPA attack based on the choice of the comparison function [MDS02]. In *all-but-nothing d-bit DPA attack* the traces are divided into three groups as follows:

$$\begin{aligned} S_{0,j} &= \{T_i | H_{i,j} = 0^d\} \\ S_{1,j} &= \{T_i | H_{i,j} = 1^d\} \\ S_{2,j} &= \{T_i | H_{i,j} \notin \{S_{0,j}, S_{1,j}\}\} \end{aligned}$$

Here $0^d$ and $1^d$ refers to the sequence of $d$ 0's and 1's respectively. Then the average traces are computed for the sets $S_{0,j}$ and $S_{1,j}$ which is used in computing $R_j$ as in the case of single-bit DPA attack. On the other hand, *generalized DPA attack* divides the traces based on the relative Hamming weight or Hamming distance of the traces as given below:

$$S_{0,j} = \{T_i | H_{i,j} \leq n - d\}$$
$$S_{1,j} = \{T_i | H_{i,j} > d\}$$
$$S_{2,j} = \{T_i | H_{i,j} \notin \{S_{0,j}, S_{1,j}\}\}$$

Here $n$ is the Hamming weight or Hamming distance of the sequence containing all 1's. The rest of the attack works similar to the all-but-nothing $d$-bit DPA attack.

**CPA attack.** In a CPA attack we determine the linear correlation between the hypothetical power values and the measured power traces. It differs from the multi-bit DPA attack only with respect to the comparison function. Here, the matrix $R$ is computed based on the correlation between the columns $H_j$ and $T_i$ (where $1 \leq j \leq K$ and $1 \leq i \leq PT$) using Pearson's correlation coefficient as follows:

$$R_{j,i} = \frac{\sum_{l=1}^{N}(H_{l,j} - \hat{H}_j).(T_{l,i} - \hat{T}_i)}{\sqrt{\sum_{l=1}^{N}(H_{l,j} - \hat{H}_j)^2 . \sum_{l=1}^{N}(T_{l,i} - \hat{T}_i)^2}}$$

Here $\hat{H}_j$ and $\hat{T}_i$ correspond to the mean values of the columns $H_j$ and $T_i$. The value $j$ corresponding to the maximum absolute value $R_{j,i}$ in matrix $R$ would be the correct key.

### 2.3.2 Template Attacks

Template attacks, contrary to other types of attacks, work in two phases. In the first phase (called *off-line phase*) we determine the leakage model of the device as opposed to the other attacks where the leakage is assumed to be following Hamming weight or Hamming distance model. In the second phase (called *on-line phase*) the actual attack is performed.

Template attacks exploit the fact that the power traces can be represented by a multivariate normal distribution defined by mean vector $m$ and covariance matrix $C$. In the off-line phase we build a template for each possible value of the intermediate variable. Namely, for every possible value of $v_i$ we compute mean vector $(m)_{v_i}$ and the covariance matrix $(C)_{v_i}$.

In the on-line phase we are given with a new trace $t$ for which we evaluate the probability density function of the normal distribution corresponding to all the templates. Namely, for every $(m, C)_{v_i}$ we compute the probability as follows:

$$p(t; (m, C)v_i) = \frac{exp(-\frac{1}{2} \cdot (t - m)' \cdot C^{-1} \cdot (t - m))}{\sqrt{(2 \cdot \pi)^T \cdot det(C)}}$$

We then determine the correct key from the template which corresponds to the highest probability (Maximum likelihood principle). Note that the templates built in the off-line phase can also be used in performing a DPA or CPA attack. Here we compute the hypothetical power values from the built model instead of using an a priori model.

### 2.3.3 Higher-order DPA Attacks

The security of a masking scheme depends on the fact that the masked variable $v_m$ and the mask $m$ are pairwise independent of each other as well as the sensitive intermediate variable $v$. If this is true, then the predicted power consumption values corresponding to the sensitive variable $v$ do not have any correlation to the actual power traces and hence it is not possible to recover the secret key. However, if we use two intermediate variables (i.e. $v_m$ and $m$) to compute the prediction values in a DPA or CPA attack, then it is possible to defeat the masking countermeasure. These types of attacks are called *second-order DPA attacks.* To prevent against second-order attacks we use second-order masking, which conceals the sensitive variable using three intermediate variables. However, second-order masking can be susceptible to third-order attacks which compute the prediction values using three shares of the sensitive variable. Indeed, this technique can be generalized to any order $d$. Namely, a $d$-th order masking scheme can be defeated by combing all the $d + 1$ shares corresponding to the intermediate variable. These attacks are called *higher-order DPA attacks.* In this section we describe the methods to perform a second-order CPA attack. The same approach can be followed in the case of higher-order CPA attacks as well.

In a second-order attack we exploit the leakages of two intermediate variables corresponding to the same mask. However, this leakage cannot be exploited directly as the intermediate variables occur at different instances in time. To counter this problem we preprocess the power traces so as to obtain the power consumption which is dependent on both the intermediate variables. Furthermore, we also need a method to compute the hypothetical values that are a combination of both the intermediate variables for which we use a *combination function.* In general, combination function depends on the type of masking used for the intermediate variable. For Boolean masking, we combine the variables using xor operation:

$$Comb(v_m, m) = v_m \oplus m = v$$

The rest of the attack works similar to a first-order CPA attack.

The preprocessing of power traces consists of two steps: selecting points of interest from the complete trace and applying appropriate preprocessing function.

**Selecting points of interest.** For a second-order attack we require two points in the power trace that correspond to the processing of the intermediate variables $v_m$ and $m$. However, in practice it is not possible to know the exact time instances at which the variables are being manipulated. On the other hand it is computationally expensive to consider all the possible pairs from a given trace. In [OMHT06], Oswald *et al.* consider all the pairs of points in a small window based on an educated guess. Later Reparaz *et al.* proposed a systematic approach to select the points of interest using Mutual Information Analysis [RGV12]. In [DSV$^{+}$14] Durvaux *et al.* proposed an alternative solution that uses projection pursuits.

**Applying preprocessing function.** Once we select the points of interest, the next step involves applying a preprocessing function on each of these points to obtain a single value. Let the leakages from a pair of points $(p_1, p_2)$ be $(L(p_1), L(p_2))$. Then, we can apply one of the following functions to obtain a preprocessed trace.

- *Absolute difference.* $pre(L(p_1), L(p_2)) = |L(p_1) - L(p_2)|$
- *Product combining.* $pre(L(p_1), L(p_2)) = L(p_1) \times L(p_2)$

If the device leaks in Hamming weight model and if we use Pearson's correlation coefficient distinguisher, *normalized product combining function* provides the best results [PRB09b]. We define the normalized product combining as follows:

$$pre(L(p_1), L(p_2)) = (L(p_1) - E[L(p_1)]) \times (L(p_2) - E[L(p_2)])$$

where $E[L(p_1)]$ and $E[L(p_2)]$ refer to the expectation of the leakages at point $p_1$ and point $p_2$ respectively.

### 2.3.4 Mutual Information Analysis

A CPA or DPA attack reveals the secret key by determining the *linear dependency* between the measured power traces and hypothetical power consumption values. To exploit all kinds of statistical dependencies (*i.e.* linear as well as non-linear) we use Mutual Information Analysis (MIA) ([BGP+11, Aum07]). MIA is based on mutual information, which gives a measure of mutual dependency between two random variables.

Mutual information between two random variables $X$ and $Y$ is formally defined as:

$$I(X, Y) = H(Y) - H(Y|X) = H(X) - H(X|Y)$$

where $H(X), H(Y)$ are the marginal entropies of $X$ and $Y$ and $H(X|Y), H(Y|X)$ are the conditional entropies. A typical MIA attack works in two stages. In the first stage, we estimate the probability density functions for different key dependent models. There exists several methods to estimate the probability density function. In particular, histograms and kernel estimation techniques have been successfully applied in the context of MIA. In the second stage, we test the dependency of the estimated models with the measured power traces. Namely, for an estimated model $M$, we test it's dependency with the leakage $L$ for every key guess $k$ by computing the mutual information between the leakage and the power trace. The key candidate that maximizes the mutual information would be the correct key.

Apart from revealing all kinds of statistical dependencies, MIA also has additional advantage that it can be applied to higher-order attacks without requiring additional preprocessing step as in the case of HO-CPA [GBPV10]. Here we estimate the multivariate probability density function involving different shares of the sensitive variable. The rest of the attack works similar to the case of first-order MIA.

### 2.3.5 Leakage Detection Tests

All the methods described so far has an important limitation: they only verify if a certain attack is possible on an implementation under chosen leakage model and the intermediate variable. As the new attacks are being discovered, it is possible that these attacks might be able to succeed although the currently known attacks are unsuccessful in recovering the secret key. Furthermore, the selection of an appropriate model is often a difficult task and error prone. Hence, there is a need for evaluation methods which are independent of the type of attacks, intermediate variables and the device leakage models. To address this problem, Gilbert *et al.* proposed two leakage assessment methodologies based on Student's t-test: *specific t-test* and *non-specific t-test* [GJJR11]. A t-test determines whether the given data sets are significantly different from each other (*i.e.* verifies the *null hypothesis*). The proposed non-specific t-test examines the device leakage without actually performing an attack. This gives an indication about the exploitable leakage in the implementation. However, note that presence of leakage does not necessarily imply a successful attack. Instead, it provides a feedback to the designers about the potential leakage sources.

In a non-specific t-test, we collect power traces corresponding to fixed and random inputs which are randomly interleaved. We divide the collected traces into two sets $S_1$ and $S_2$ corresponding to the fixed and random inputs respectively. Then, we apply Welsch's t-test to each point in the collected trace. For a single point on the trace, t-test is computed as:

$$t = \frac{\mu_1 - \mu_2}{\sqrt{\frac{V_1}{|S_1|} + \frac{V_2}{|S_2|}}}$$

where $\mu_i, V_i, |S_i|$ denote mean, variance and number of traces for the set $S_i$. Based on the t-value corresponding to fixed and random input sets, we determine the presence of first-order leakage. Typically for $|t| > 4.5$, a first-order attack is highly feasible.

## 2.4 Boolean Masking

In Boolean masking each sensitive variable $x$ is represented by two Boolean shares $x_1, x_2$ so that $x = x_1 \oplus x_2$. Block ciphers make repeated use of key-dependent transformations called *round transformations*. The *round transformations* in turn are a combination of linear and non-linear functions. To protect block ciphers against side-channel attacks using masking, we need to mask both these functions. In general, masking a linear function is easy since:

$$f(x_1 \oplus x_2) = f(x_1) \oplus f(x_2)$$

which makes it easy to evaluate the function $f$ on the two shares independently. However, the non-linear functions (*e.g.* S-box) are difficult to mask as they don't provide such an easy relationship. In general non-linear functions are masked using two methods:

1. Write the non-linear function as a combination of linear/affine functions and protect them independently (*e.g.* [OMPR05, AG01, BGK05])
2. Use a randomized lookup table to store the masked values of the function outputs (*e.g.* [HOM06]) [2]

The same approach can be used for higher-order masking as well.

### 2.4.1 Generic Countermeasure Against Second-order DPA

At FSE 2008, Rivain et al proposed two algorithms to protect the computation of S-box outputs against second-order attacks [RDP08]. Given three input shares of a secret value $x$, namely $x_1 = x \oplus x_2 \oplus x_3$, $x_2$, and $x_3$ (which are all in $\mathbb{F}_{2^n}$) and two output shares $y_1, y_2 \in \mathbb{F}_{2^m}$ along with an $(n, m)$ S-box lookup function $S$, they compute the third share $y_3$ such that $y_1 \oplus y_2 \oplus y_3 = S(x)$. Hence, we have $y_3 = S(x) \oplus y_1 \oplus y_2$. The algorithms compute $(S(x_1 \oplus a) \oplus y_1) \oplus y_2$ for all possible values of $a$ (i.e. $0 \leq a \leq 2^{n-1}$), among which the correct value can be obtained when $a = x_2 \oplus x_3$. We recall these algorithms below.

**Algorithm 1** Prouff-Rivain Sec2O-masking: First Variant

```
Input: Three input shares: $(x_1 = x \oplus x_2 \oplus x_3, x_2, x_3) \in \mathbb{F}_{2^n}$, two output shares:
    $(y_1, y_2) \in \mathbb{F}_{2^m}$, and an $(n, m)$ S-box lookup function $S$
Output: Masked S-box output: $S(x) \oplus y_1 \oplus y_2$
1: Randomly generate $n$-bit number $r$
2: $r' \leftarrow (r \oplus x_2) \oplus x_3$
3: for $a = 0$ to $2^n - 1$ do
4:     $a' \leftarrow a \oplus r'$
5:     $T[a'] \leftarrow ((S(x_1 \oplus a) \oplus y_1) \oplus y_2)$
6: end for
7: return $T[r]$
```

Algorithm 1 uses a table of $2^n$ entries to store $(S(x_1 \oplus a) \oplus y_1) \oplus y_2$ for all possible values of $a$. Here, the value $(x_2 \oplus x_3)$ is masked via a random variable $r$, the result of which is assigned to $r'$. Thereafter, the entry corresponding to $(S(x_1 \oplus a) \oplus y_1) \oplus y_2$ will be stored at location $a' = a \oplus r'$. The correct value of the third share $y_3$ can be retrieved by accessing the value stored in the table at location $T[r]$. As $r = a'$, the value of $a$ becomes $a = r \oplus r' = x_2 \oplus x_3$, thus yielding the desired result.

The security of Algorithm 1 can be proven by showing that it is impossible to recover $x$ by combining any pair of intermediate variables computed by the algorithm. Please refer to Section 3.1 in [RDP08] for the complete proof. Algorithm 1 requires a table of $2^n$ words (each having a length of $m$ bits) in RAM, and is, therefore, not suitable for low-cost devices. To overcome this issue, Rivain et al introduced another algorithm consuming less memory at the expense of executing more operations.

[2] In several cases S-boxes are implemented using lookup tables.

**Algorithm 2** Prouff-Rivain Sec2O-masking: Second Variant

```
Input: Three input shares: $(x_1 = x \oplus x_2 \oplus x_3, x_2, x_3) \in \mathbb{F}_{2^n}$, two output shares:
    $(y_1, y_2) \in \mathbb{F}_{2^m}$, and an $(n, m)$ S-box lookup function $S$
Output: Masked S-box output: $S(x) \oplus y_1 \oplus y_2$
1: Randomly generate one bit $b$
2: for $a = 0$ to $2^n - 1$ do
3:     $cmp \leftarrow compare_b(x_2 \oplus a, x_3)$
4:     $R_{cmp} \leftarrow ((S(x_1 \oplus a) \oplus y_1) \oplus y_2)$
5: end for
6: return $R_b$
```

Algorithm 2 gives the second solution proposed by Rivain et al in [RDP08] to securely compute an S-box output. In this variant, they use a function called $compare_b(x, y)$, which returns $b$ if $x = y$ and $\bar{b}$ otherwise. A first-order secure implementation of $compare_b$ is necessary to guarantee the security of the algorithm. To this end, Rivain et al [RDP08] presented a method for implementing the $compare_b$ function, recalled in Algorithm 3. The secure S-box computation works as follows: First, a random bit $b$ is generated, which is one of the inputs to the $compare_b$ function. Then, for each possible value of $a$, the algorithm computes $(S(x_1 \oplus a) \oplus y_1) \oplus y_2$, which will be written to either $R_b$ or $R_{\bar{b}}$, depending on the actual output of the $compare_b$ function. The inputs to the $compare_b$ function are $x_2 \oplus a$ and $x_3$. When $a = x_2 \oplus x_3$, $compare_b(x_2 \oplus a, x_3)$ returns $b$, thus the result is stored in $R_b$. In all other cases, the returned value is $\bar{b}$, so the result is stored in the register $R_{\bar{b}}$. At the end of the algorithm, the value stored in $R_b$ is $S(x) \oplus y_1 \oplus y_2$, which is exactly what we wanted to achieve.

Note that Algorithm 2 needs only $2^n$ bits in RAM, namely for the function $compare_b$. Thus, it requires $m$ times less memory than Algorithm 1, though the execution time is longer due to multiple calls to the $compare_b$ function.

**Algorithm 3** Compare Function

```
Input: $x, y, b, n$
Output: $b$ if $x = y$, $\bar{b}$ otherwise
1: $r_3 \leftarrow rand(n)$
2: $T[0 : 2^n - 1] \leftarrow \bar{b}, \bar{b}, \ldots, \bar{b}$
3: $T[r_3] \leftarrow b$
4: return $T[(x \oplus r_3) \oplus y]$
```

### 2.4.2 Higher-order Masking

The first higher-order resistant implementation of AES was proposed by Schramm and Paar [SP06]. Given $n$ shares of the sensitive variable $x$: $(x_1 = x \oplus x_1 \oplus \cdots \oplus x_n, x_2, \cdots, x_n)$ they compute the $n$ shares of $S(x)$: $(S(x) \oplus y_1 \oplus \cdots \oplus y_n, y_1, \cdots, y_n)$ using repeated recomputation of the masked table. We recall their first algorithm in Algorithm 4.

**Algorithm 4** Schramm-Paar HO-masking

```
Input: $n$ input shares: $(x_1, x_2, \cdots, x_n)$, $n-1$ output shares: $(y_1, \cdots, y_n)$, and an
    $(n, m)$ S-box lookup function $S$
Output: Masked S-box output: $S(x) \oplus y_1 \oplus \cdots \oplus y_n$
1: for $i = 1$ to $n$ do
2:     for $a = 0$ to $2^n - 1$ do
3:         $S^*[a] \leftarrow S[a \oplus x_i] \oplus y_i$
4:     end for
5:     $S \leftarrow S^*$
6: end for
7: return $S(x_1)$
```

The above algorithm recomputes the lookup table $n$ times (for security against attacks of order $n-1$) for each round of the cipher. This makes the implementation significantly slower. To improve the performance, they propose a modification, which performs $n$ recomputations only for the first S-box in the first round. For the rest of the S-box computations they recompute the table only once. However, Coron *et al.* showed that this scheme is insecure against attacks of order greater than 2 [CPR07]. This has been recently fixed by Coron, who proposed a higher-order secure algorithm to compute randomized lookup table [Cor14].

On the other hand, solutions also exist for higher-order secure computation of S-box on the fly. At CHES 2010, Rivain and Prouff [RP10] proposed an algorithm to protect the AES against $n$-th order attacks based on the Ishai-Sahai-Wagner construction [ISW03]. Their basic idea is to write the AES round transformations as operations in the field GF($2^8$) and mask additions and multiplications. This approach can be extended to any S-box by considering the polynomial representation of the S-box, which can be computed using Lagrange polynomial interpolation over a finite field [CGP+12a]. This was later improved by Roy, Vivek in [RV13] and Coron, Roy, Vivek in [CRV14].

## 2.5 Goubin's Conversion Algorithms

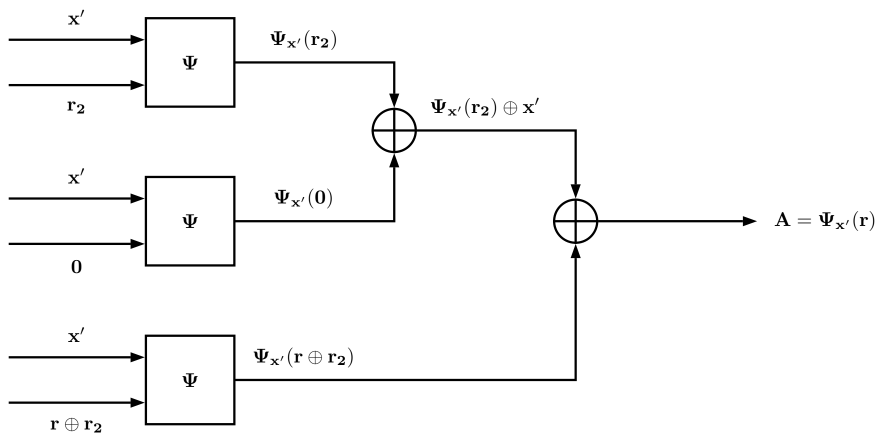
In this section we recall Goubin's algorithm for converting from Boolean masking to arithmetic masking and conversely [Gou01], secure against first-order attacks. Given a $k$-bit variable $x$, for Boolean masking we write:

$$x = x' \oplus r$$

where $x'$ is the masked variable and $r \leftarrow \{0,1\}^k$. Similarly for arithmetic masking we write

$$x = A + r \bmod 2^k$$

In the following all additions and subtractions are done modulo $2^k$, for some parameter $k$.



$\mathbf{x'}$
$\mathbf{r_2}$
$\Psi$
$\Psi_{\mathbf{x'}}(\mathbf{r_2})$
$\Psi_{\mathbf{x'}}(\mathbf{r_2}) \oplus \mathbf{x'}$
$\mathbf{x'}$
$\mathbf{0}$
$\Psi$
$\Psi_{\mathbf{x'}}(\mathbf{0})$
$\mathbf{A} = \Psi_{\mathbf{x'}}(\mathbf{r})$
$\mathbf{x'}$
$\mathbf{r} \oplus \mathbf{r_2}$
$\Psi$
$\Psi_{\mathbf{x'}}(\mathbf{r} \oplus \mathbf{r_2})$

Figure 2.2: Goubin Boolean to arithmetic conversion

## 2.5.1 Boolean to Arithmetic Conversion

We first recall the Boolean to arithmetic conversion method from Goubin [Gou01]. One considers the following function $\Psi_{x'}(r) : \mathbb{F}_{2^k} \to \mathbb{F}_{2^k}$:

$$\Psi_{x'}(r) = (x' \oplus r) - r$$

**Theorem 2.2** (Goubin [Gou01]). *The function $\Psi_{x'}(r) = (x' \oplus r) - r$ is affine over $\mathbb{F}_2$.*

Using the affine property mentioned above, the conversion from Boolean to arithmetic masking is straightforward. Given $x', r \in \mathbb{F}_{2^k}$ we must compute $A$ such that $x' \oplus r = A + r$. From the affine property of $\Psi_{x'}(r)$ we can write:

$$A = (x' \oplus r) - r = \Psi_{x'}(r) = \big(\Psi_{x'}(r_2) \oplus \Psi_{x'}(0)\big) \oplus \Psi_{x'}(r \oplus r_2)$$

for any $r_2 \in \mathbb{F}_{2^k}$. Therefore the technique consists in first generating a uniformly distributed random $r_2$ in $\mathbb{F}_{2^k}$, then computing $\Psi_{x'}(r \oplus r_2)$ and $\Psi_{x'}(r_2) \oplus \Psi_{x'}(0)$ separately, and finally performing XOR operation on these two to get $A$ (refer to Figure 2.2). The technique is clearly secure against first-order attacks; namely the left term $\Psi_{x'}(r \oplus r_2)$ is independent from $r$ and therefore from $x = x' \oplus r$, and the right term $\Psi_{x'}(r_2) \oplus \Psi_{x'}(0)$ is also independent from $r$ and therefore from $x$. Note that the technique is very efficient as it requires only a constant number of operations (independent of $k$). We recall the full algorithm in Algorithm 5.

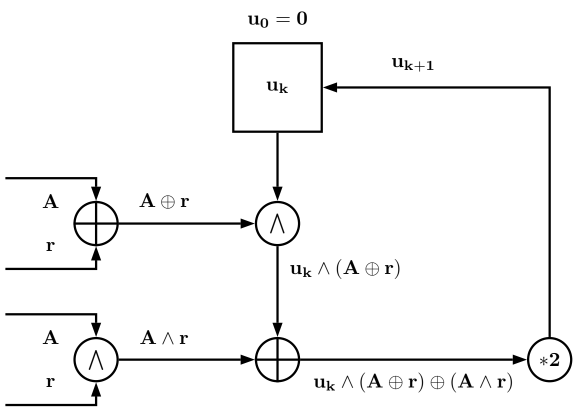


Figure 2.3: Goubin arithmetic to Boolean conversion

**Algorithm 5** Goubin B→A Conversion

```
Input: $x', r$ such that $x' = x \oplus r$
Output: $A, r$ such that $x = A + r$
1: $r_2 \leftarrow \mathsf{rand}(k)$
2: $T \leftarrow x' \oplus r_2$
3: $T \leftarrow T - r_2$
4: $T \leftarrow x' \oplus T$
5: $S \leftarrow r \oplus r_2$
6: $A \leftarrow S \oplus x'$
7: $A \leftarrow A - S$
8: $A \leftarrow A \oplus T$
```

### 2.5.2 From Arithmetic to Boolean Masking

Goubin also described in [Gou01] a technique for converting from arithmetic to Boolean masking, secure against first-order attacks. However it is more complex than from Boolean to arithmetic masking; its complexity is $\mathcal{O}(k)$ for additions modulo $2^k$. It is based on the following theorem.

**Theorem 2.3** (Goubin [Gou01]). *If we denote $x' = (A + r) \oplus r$, we also have $x' = A \oplus u_{k-1}$, where $u_{k-1}$ is obtained from the following recursion formula:*

$$\begin{cases} u_0 = 0 \\ \forall k \geq 0, u_{k+1} = 2[u_k \wedge (A \oplus r) \oplus (A \wedge r)] \end{cases} \tag{2.1}$$

The recursion formula is shown pictorially in Figure 2.3. From Theorem 2.3, one obtains the following corollary.

**Corollary 2.1** ([Gou01]). *For any random $\gamma \in \mathbb{F}_{2^k}$, if we assume $x' = (A + r) \oplus r$, we also have $x' = A \oplus 2\gamma \oplus t_{k-1}$, where $t_{k-1}$ can be obtained from the following* *recursion formula:*

$$\begin{cases} t_0 = 2\gamma \\ \forall i \geq 0, t_{i+1} = 2[t_i \wedge (A \oplus r) \oplus \omega] \end{cases} \tag{2.2}$$

*where* $\omega = \gamma \oplus (2\gamma) \wedge (A \oplus r) \oplus A \wedge r$.

Since the iterative computation of $t_i$ contains only XOR and AND operations, it can easily be protected against first-order attacks. This gives the algorithm below.

**Algorithm 6** Goubin A→B Conversion

```
Input: $A, r$ such that $x = A + r$
Output: $x', r$ such that $x' = x \oplus r$
 1: $\gamma \leftarrow \mathsf{rand}(k)$
 2: $T \leftarrow 2\gamma$
 3: $x' \leftarrow \gamma \oplus r$
 4: $\Omega \leftarrow \gamma \wedge x'$
 5: $x' \leftarrow T \oplus A$
 6: $\gamma \leftarrow \gamma \oplus x'$
 7: $\gamma \leftarrow \gamma \wedge r$
 8: $\Omega \leftarrow \Omega \oplus \gamma$
 9: $\gamma \leftarrow T \wedge A$
10: $\Omega \leftarrow \Omega \oplus \gamma$
11: for $j := 1$ to $k - 1$ do
12:     $\gamma \leftarrow T \wedge r$
13:     $\gamma \leftarrow \gamma \oplus \Omega$
14:     $T \leftarrow T \wedge A$
15:     $\gamma \leftarrow \gamma \oplus T$
16:     $T \leftarrow 2\gamma$
17: end for
18: $x' \leftarrow x' \oplus T$
19: return $x'$
```

We can see that the total number of operations in the above algorithm is $5k+5$, in addition to one random number generation. Karroumi *et al.* recently improved Goubin's conversion scheme down to $5k+1$ operations [KRJ04]. More precisely they start the loop in (2.2) from $i = 2$ instead of $i = 1$, and compute $t_1$ directly with a single operation, which decreases the number of operations by 4.

Karroumi *et al.* also provided an algorithm to compute first-order secure addition on Boolean shares using Goubin's recursion formula, requiring $5k+8$ operations and a single random generation. More precisely, given two sensitive variables $x$ and $y$ masked as $x = x' \oplus s$ and $y = y' \oplus r$, their algorithm computes two shares $z_1 = (x + y) \oplus r \oplus s, z_2 = r \oplus s$ using Goubin's recursion formula (2.1); we recall their solution in Algorithm 7.

**Algorithm 7** KRJ SecAdd

```
Input: $x', s, y', r$ such that $x = x' \oplus s$ and $y = y' \oplus r$
Output: $z_1, z_2$ such that $z_1 \oplus z_2 = (x + y)$
 1: $\gamma \leftarrow \mathsf{rand}(k)$
 2: $C \leftarrow \gamma$
 3: $T \leftarrow x' \wedge y';\ \Omega \leftarrow C \oplus T$
 4: $T \leftarrow x' \wedge r;\ \Omega \leftarrow \Omega \oplus T$
 5: $T \leftarrow y' \wedge s;\ \Omega \leftarrow \Omega \oplus T$
 6: $T \leftarrow r \wedge s;\ \Omega \leftarrow \Omega \oplus T$
 7: $B \leftarrow \Omega \ll 1;\ C \leftarrow C \ll 1$
 8: $A_0 \leftarrow x' \oplus y'; A_1 \leftarrow r \oplus s$
 9: $T \leftarrow C \wedge A_0;\ \Omega \leftarrow \Omega \oplus T$
10: $T \leftarrow C \wedge A_1;\ \Omega \leftarrow \Omega \oplus T$
11: for $j := 2$ to $k - 1$ do
12:     $T \leftarrow B \wedge A_0$
13:     $B \leftarrow B \wedge A_1$
14:     $B \leftarrow B \oplus \Omega$
15:     $B \leftarrow B \oplus T$
16:     $B \leftarrow B \ll 1$
17: end for
18: $A_0 \leftarrow A_0 \oplus B$
19: $A_0 \leftarrow A_0 \oplus C$
20: return $(A_0, A_1)$
```

## 2.6 Conversion Algorithms Based on Lookup Tables

The arithmetic to Boolean conversion algorithm proposed by Goubin requires $5k+5$ operations when the size of the conversion operand is $k$ bits. Several algorithms were later proposed to improve the efficiency using lookup tables. In this section we recall two of them: Coron-Tchulkine method [CT03] and Debraize method [Deb12]. A similar method was also was proposed in [NP04] in the context of protecting IDEA block cipher.

### 2.6.1 Coron-Tchulkine Algorithm

The first solution to the problem of arithmetic to Boolean conversion using lookup tables was given by Coron and Tchulkine in [CT03]. To convert arithmetic shares of size $k$ bits to corresponding Boolean shares, their first method uses a table of $2^k$ bits. We recall their algorithm to create the lookup table in Algorithm 8.

**Algorithm 8** Coron-Tchulkine A→B Conversion: LUT

```
Input:
Output: Table $T$ and used random number $r$
1: $r \leftarrow \mathsf{rand}(k)$
2: for $a := 0$ to $2^k - 1$ do
3:     $T[a] \leftarrow (a + r) \oplus r$
4: end for
5: return $T, r$
```

For converting an arithmetic share $A = x - r$ to a Boolean share $x' = x \oplus r$, we can just access the table entry $T[A] = (A + r) \oplus r = x \oplus r$. This requires only constant time as in case of Goubin's Boolean to arithmetic conversion. However, this solution is not practical for larger $k$ (*e.g.* $k > 10$) as we need a lookup table of $2^k$ entries. To overcome this challenge, they proposed a solution using two $l$-bit tables, where $l$ is the size of the lookup table (generally $l < 8$) and $k = l \cdot p$ for some $p > 0$.

Let us assume that the sensitive variable $x$ is represented by two arithmetic shares $A$ and $R$ so that $x = A + R \mod 2^k$. We need to obtain corresponding Boolean shares $x', r$ such that $x = x' \oplus R$. Their solution works based on recursively applying $l$-bit lookup table to $l$-bit words starting from the least significant word. Let $A = A_1||A_2$, $R = R_1||R_2$ and $x = x_1||x_2$ where $A_1, R_1, x_1$ are of $(p-1) \cdot l$ bits and $A_2, R_2, x_2$ are of $l$ bits. For a random $l$-bit integer $r$ let

$$A = (A - r) + R_2 \mod 2^k$$

That implies

$$x = (A_1||A_2) + (R_1||r) \mod 2^k$$

which can be written as

$$x_1 = A_1 + R_1 + c \mod 2^{(p-1)\cdot l}$$
$$x_2 = A_2 + r \mod 2^l$$

where $c$ is the carry from the addition $A_2+r$ which also need to be protected against first-order attacks. To that effect they propose to use another $l$-bit table for storing the carry (recalled in Algorithm 9).

**Algorithm 9** Coron-Tchulkine A→B Conversion: Carry LUT

```
Input:
Output: Table $C$ and used random number $s$
1: $s \leftarrow \mathsf{rand}(k)$
2: for $a := 0$ to $2^k - 1$ do
3:     $C[a] \leftarrow s + \mathsf{carry}(a, r) \mod 2^{(p-1)\cdot l}$        ▷ $\mathsf{carry}(a, b)$ returns the carry from $a + b$
4: end for
5: return $C, s$
```

We can then compute $A_1$ using the following formula:

$$A_1 = A_1 + C[A_2] - s \mod 2^{(p-1)\cdot l}$$

Now $x'_2$ can be directly computed using the lookup table created by Algorithm 8. Namely,

$$\begin{aligned} x'_2 &= (T[A_2] \oplus R_2) \oplus r_2 \\ &= (x_2 \oplus r_2 \oplus R_2) \oplus r_2 \\ &= x_2 \oplus r_2 \end{aligned}$$

This gives the last $l$-bits of the Boolean share $x'$. The same process can be applied to $A_1, R_1$ to get the next $l$-bits of $x'$ and by recursive application we can obtain the full $k$-bits of $x'$. We recall the full algorithm in Algorithm 10.

**Algorithm 10** Coron-Tchulkine A→B Conversion

```
Input: Operand size $k$, arithmetic shares $A, R$ such that $x = A + R \mod 2^k$, Tables $T, C$,
    random numbers $r, s$ from Algorithm 8 and Algorithm 9
Output: $x', R$ such that $x = x' \oplus R$
 1: $A \leftarrow A - r \mod 2^k$
 2: $A \leftarrow A + R_2 \mod 2^k$        ▷ $R = R_1||R_2$, $R_1 : ((p-1)\cdot l)$-bit and $R_2$ is $l$-bit
 3: if $p = 0$ then
 4:     $x' \leftarrow (T[A] \oplus R_2) \oplus r$
 5:     return $x'$
 6: else
 7:     $A \leftarrow A_1||A_2$
 8: end if
 9: $A_1 \leftarrow A_1 + C[A_2] \mod 2^{(p-1)\cdot l}$
10: $A_1 \leftarrow A_1 - s \mod 2^{(p-1)\cdot l}$
11: $x'_2 \leftarrow (T[A_2] \oplus R_2) \oplus r$
12: $x'_1 \leftarrow$ Coron A→B Conversion $((p-1)\cdot l, A_1, R_1, T, C, r, s)$
13: $x' \leftarrow x'_1||x'_2$
14: return $x'$
```

**Flaw in Coron-Tchulkine Method** The above algorithm doesn't work as expected in certain cases as shown by Blandine Debraize in [Deb12]. Assume that following conditions hold simultaneously:

1. $p > 2$
2. $s = 2^l$
3. carry in step 9 is 1

It implies that in the initial call to the conversion function, we have:

$$\begin{aligned} A_1 &> 2^l \\ A_1 + s \mod 2^{(p-1)\cdot l} &< A_1 \\ A_1 + C[A_2] - s &\neq A_1 + 1 \end{aligned}$$

which is not the expected result. Debraize also provided a fix to the flaw that makes the algorithm significantly slower. To overcome this limitation, a new method was proposed in the same paper which is recalled below.

### 2.6.2 Debraize Solution

In this subsection we recall the algorithm proposed by Debraize to efficiently convert from arithmetic to Boolean masking secure against first-order attacks[Deb12]. In contrast to the earlier methods which use arithmetic masks, carries here are protected by Boolean masks. The two tables from Coron-Tchulkine's algorithm are combined into one. The combined Table $T$ has entries for both the carry and no carry cases and are differentiated by a random bit $\rho$. The entry for the no carry case for input $a$ and random number $r$ is given as

$$T[\rho||a] = (a + r) \oplus (\rho||r)$$

and for the carry case it is given as

$$T[(\rho \oplus 1)||a] = (a + r + 1) \oplus (\rho||r).$$

Namely, in case of carry the table entry corresponding to the carry would be $\rho + 1$ and $\rho$ for the no carry case. The algorithm to create the lookup table is recalled in Algorithm 11.

**Algorithm 11** Debraize Table Creation

```
Input:
Output: Table $T$, $r$, $\rho$
1: $r \leftarrow \mathsf{rand}(k)$
2: $\rho \leftarrow \mathsf{rand}(1)$
3: for $a = 0$ to $2^k - 1$ do
4:     $T[\rho||a] \leftarrow (A + r) \oplus (\rho||r)$
5:     $T[(\rho \oplus 1)||a] \leftarrow (A + r + 1) \oplus (\rho||r)$
6: end for
7: return $T, r, \rho$
```

Similar to Coron-Tchulkine method, the input arithmetic share is processed in words of $l$ bits. For $i^{th}$ arithmetic word $A^i$, the corresponding Boolean share $x_1^i$ is retrieved from the table by accessing the entry $(\beta||(A^i - r) + R^i)$, and later XORing with $R^i$ and $r$. Here $\beta$ is the Boolean mask of the carry. Initially $\beta$ is set to $\rho$ and after conversion of every word, it is modified according to the carry arising from that word. We recall the full algorithm in Algorithm 12.

**Algorithm 12** Debraize A→B Conversion

```
Input: Arithmetic shares: $A, R$ and $r, \rho$ from precomputed table
Output: Boolean shares: $x_1, x_2$
 1: $A \leftarrow A - (r||\cdots||r) \mod 2^n$
 2: $\beta \leftarrow \rho$
 3: for $i = 0$ to $p - 1$ do
 4:     Split $A$ into $A_h||A_l$ and $R$ into $R_h||R_l$ such that $A_l$ and $R_l$ of size $k$
 5:     $A \leftarrow A + R_l \mod 2^{(n-i)\cdot k}$
 6:     $\beta||x_{1_i} \leftarrow T[\beta||A_l]$
 7:     $x_{1^i} \leftarrow x_{1^i} \oplus R_l$
 8:     $A \leftarrow A_h$
 9:     $R \leftarrow R_h$
10: end for
11: $x_1 = (x_1^0||x_1^1||\cdots||x_1^i) \oplus (r||r||\cdots||r)$
12: $x_2 = R$
13: return $x_1, x_2$
```

# Chapter 3

# Secure Conversion between Boolean and Arithmetic Masking of Any Order

The current solutions to convert between Boolean and arithmetic masking are secure against only first-order attacks. This chapter presents and evaluates new conversion algorithms that are secure against attacks of any order. To set the context, we show that a straightforward extension of first-order conversion schemes to second order is not possible. Then we introduce our new algorithms to convert between Boolean and arithmetic masking. To convert masks of a size of $k$ bits securely against attacks of order $n$, the proposed algorithms have a time complexity of $\mathcal{O}(n^2k)$ in both directions and are proven to be secure in the Ishai, Sahai, and Wagner (ISW) framework for private circuits. We evaluate our algorithms using HMAC-SHA-1 as example and report the execution times we achieved on a 32-bit AVR microcontroller. This is a joint work worth Jean-Sébastien Coron and Johann Großschädl. A part of this work appeared in the proceedings of CHES 2014 [CGV14].

**Contents**

## 3.1 Introduction

**Security Model.**

We definitely aim for countermeasures against side-channel attacks that can be proven secure in a reasonable model of side-channel leakage (i.e. we will not be satisfied with heuristic "ad-hoc" countermeasures). Perhaps the simplest such model is the probing attack model proposed by Ishai, Sahai and Wagner (ISW) at CRYPTO 2003 [ISW03] (see Section 3.3). They initiated the theoretical study of securing circuits against an adversary who can probe its wires. In this model, the attacker is allowed to access at most $t$ wires of the circuit, but he should not be able to learn anything about the secret key. The authors show that any circuit $C$ can be transformed into a new circuit of size $\mathcal{O}(t^2 \cdot |C|)$ that is resistant against such an adversary. The approach is based on secret-sharing every variable $x$ into $n$ shares $x_i$ with $x = x_1 \oplus x_2 \cdots \oplus x_n$, and processing the shares in a way so that no information about the initial variable $x$ can be learned by any $t$-limited adversary, for $n \geq 2t + 1$.

In recent years, numerous papers on provable security against side-channel attacks have been published in the literature, forming the rapidly emerging field of leakage-resilient cryptography. Building upon the leakage model introduced by Micali and Reyzin [MR04] and on the bounded retrieval model [CLW06, Dzi06], the leakage resilience model assumes that the adversary has the ability to repeatedly learn arbitrary functions of the secret key, as long as the total number of bits leaked to the adversary is bounded by some parameter $L$. This is a very strong security notion because an attacker can choose arbitrary leakage functions; only the amount of leaked information is bounded. In particular, it is more general than the ISW probing model [ISW03], in which the attacker has only access to a limited number of physical bits computed in the circuit.

However, cryptosystems proven secure in the most general leakage-resilient model are often too inefficient for practical use. In practice, one typically has to design a countermeasure against side-channel attacks for an existing algorithm (such as AES or HMAC-SHA-1) instead of devising a completely new algorithm based on the principles of leakage-resilient cryptography. The main advantage of the ISW probing model is that it can potentially lead to relatively practical designs. Another benefit is its interplay with resistance against power analysis attacks. Namely, if a given algorithm is proven resistant against $t$ probes in the ISW model, then (at least) $t + 1$ measurements in a power acquisition must be combined to obtain the key. As shown in [CJRR99], the number of power acquisitions required to recover the key grows exponentially with $t$. This means that, even if a real probing attack would be physically impossible or too costly, it makes sense to obtain countermeasures with the largest possible value of $t$ since this translates into an (exponentially in $t$) increasing level of security against power attacks. In this chapter, we mainly work in the ISW model.

Proving the resistance of a countermeasure against a single-probe attack (or a first-order attack) is usually straightforward since it suffices to show that all intermediate variables are uniformly distributed (or, at least, that their distribution is independent from the secret key) as in this case a single probe reveals no information to the attacker. To prove resistance against $t$ probes, one should *a priori* consider every possible $t$-tuple of variables and show that their joint distribution is independent from the secret key. This approach has been used to prove the security of algorithms against second-order attacks [RDP08]. However, as the number of such $t$-tuples grows exponentially with $t$, this analysis becomes unfeasible, even for small values of $t$. To work around this problem, Ishai, Sahai and Wagner introduced in [ISW03] a very practical simulation framework in which one shows how to simulate any set of $t$ wires probed by the adversary from a subset of the input shares of the transformed circuits. Since any proper subset of these input shares can be simulated without knowledge of the input values in the original circuit, a perfect simulation of the $t$ probed wires is possible. We follow the same approach in this chapter.

#### Our Contribution.

Currently, there exists no practical conversion technique that works for masking of order two or higher. The present chapter attempts to fill this gap. We introduce the first conversion algorithms between Boolean and arithmetic masking that are secure against $t$-th order attacks (instead of first-order only). We start with the problem of how to apply arithmetic operations directly on Boolean shares and present an algorithm for secure addition modulo $2^k$ with $n$ shares (where $n \geq 2t+1$) that has a complexity of $\mathcal{O}(n^2k)$. Then, we introduce algorithms to convert from Boolean to arithmetic masking and vice versa, again with a complexity of $\mathcal{O}(n^2k)$ in both directions. These algorithms are proven secure in the Ishai, Sahai and Wagner (ISW) framework for private circuits [ISW03].

We apply our countermeasures to protect HMAC-SHA-1 against second and third-order attacks. We implemented and evaluated all our masking schemes on a 32-bit AVR processor. Based on a detailed performance analysis, we identify the most efficient algorithms in practice for different levels of security.

#### Organization.

We first show that a straightforward application of Goubin's method is insecure for second-order conversion in Section 3.2. We then briefly recall the ISW framework that we use to prove our algorithms in Section 3.3. Our solutions to securely perform addition on Boolean shares are given in Section 3.4. The algorithms for secure conversion from arithmetic to Boolean masking and Boolean to arithmetic masking are presented in Section 3.5 and Section 3.6 respectively. Section 3.7 gives the implementation results on a 32-bit microcontroller.

## 3.2 Applying Goubin's Conversion to Second Order

In this section, we demonstrate that a straightforward application of Goubin's conversion technique [Gou01] to the second order does not work. Assume we have three Boolean shares $x_1$, $x_2$, $x_3$ whereby $x = x_1 \oplus x_2 \oplus x_3$. We need to find the arithmetic shares $A_1$, $A_2$, and $A_3$ such that $x = A_1 + A_2 + A_3 \bmod 2^n$. To do so, we can iteratively compute $A_1$, $A_2$, $A_3$ as follows:

$$\begin{aligned} x &= A_1 + (x_2 \oplus x_3) \\ x &= A_1 + A_2 + x_3 \\ A_1 &= x - (x_2 \oplus x_3) \\ A_2 &= (x_2 \oplus x_3) - x_3 \\ A_3 &= x_3 \end{aligned}$$

Based on the above, we can compute $A_1$ in the following way:

$$\begin{aligned} A_1 &= x_1 \oplus (x_2 \oplus x_3) - (x_2 \oplus x_3) = \phi_{x_1}(x_2 \oplus x_3) \\ &= \phi_{x_1}(x_2) \oplus \phi_{x_1}(x_3) \oplus x_1. \end{aligned}$$

One could try to securely compute $\phi_{x_1}(x_2)$ and $\phi_{x_1}(x_3)$ as follows:

$$\begin{aligned} \phi_{x_1}(x_2) &= \phi_{x_1}(x_2 \oplus r) \oplus \phi_{x_1}(r) \oplus x_1 \\ \phi_{x_1}(x_3) &= \phi_{x_1}(x_3 \oplus r) \oplus \phi_{x_1}(r) \oplus x_1. \end{aligned}$$

This means,

$$\begin{aligned} A_1 &= \phi_{x_1}(x_2 \oplus r) \oplus \phi_{x_1}(r) \oplus \phi_{x_1}(x_3 \oplus r) \oplus \phi_{x_1}(r) \oplus x_1 \\ &= \phi_{x_1}(x_2 \oplus r) \oplus \phi_{x_1}(x_3 \oplus r) \oplus x_1 \\ &= ((x_1 \oplus x_2 \oplus r) - (x_2 \oplus r)) \oplus ((x_1 \oplus x_3 \oplus r) - (x_3 \oplus r)) \oplus x_1. \end{aligned}$$

But we can combine the leakages from $x_1 \oplus x_2 \oplus r$ and $x_3 \oplus r$ to get $x_1 \oplus x_2 \oplus x_3 = x$, inducing a second order attack. Similarly, we can combine the leakages from $x_1 \oplus x_3 \oplus r$ and $x_2 \oplus r$ to get $x_1 \oplus x_2 \oplus x_3 = x$. Now, let us consider the case where we use a different random $r_i$ for computing each $\phi_{x_i}(x_j)$, i.e.

$$\begin{aligned} \phi_{x_1}(x_2) &= \phi_{x_1}(x_2 \oplus r_1) \oplus \phi_{x_1}(r_1) \oplus x_1 \\ \phi_{x_1}(x_3) &= \phi_{x_1}(x_3 \oplus r_2) \oplus \phi_{x_1}(r_2) \oplus x_1. \end{aligned}$$

This means,

$$A_1 = \phi_{x_1}(x_2 \oplus r_1) \oplus \phi_{x_1}(r_1) \oplus \phi_{x_1}(x_3 \oplus r_2) \oplus \phi_{x_1}(r_2) \oplus x_1.$$

Now, when computing $A_1$, regardless of what sequence we choose, we would be leaking the secret $x$ while combining the results. For example, assume that we calculate according to the following sequence:

$$\begin{aligned} \phi_{x_1}(r_1) \oplus \phi_{x_1}(x_3 \oplus r_2) \oplus \phi_{x_1}(r_2) &= \phi_{x_1}(x_3 \oplus r_1) \\ &= ((x_1 \oplus x_3 \oplus r_1) - (x_3 \oplus r_1)) \end{aligned}$$

Let us further assume that $\phi_{x_1}(x_2 \oplus r_1)$ is calculated as follows:

$$\phi_{x_1}(x_2 \oplus r_1) = ((x_1 \oplus x_2 \oplus r_1) - (x_2 \oplus r_1))$$

Then, we can combine the leakages from $x_1 \oplus x_2 \oplus r_1$ and $x_3 \oplus r_1$ to find the value of $x$. From this, we conclude that the straightforward application of the method of Goubin does not work for second order.

## 3.3 The Ishai, Sahai and Wagner Framework

In this section we describe the framework of Ishai, Sahai and Wagner (ISW) [ISW03] for proving the security against an adversary observing at most $t$ variables within a circuit. We will use this framework in Section 3.5 and in Section 3.6 to prove the security of our conversion algorithms.

A *stateless* circuit over $\mathbb{F}_2$ can be defined as a directed acyclic graph whose sources and sinks are input and output variables, respectively, while its vertices are Boolean gates [Cor14]. Such a stateless circuit can be augmented with random-bit gates to form a *randomized circuit*. As stated in [ISW03], a *random-bit gate* has no input and produces as output a uniformly random bit at each new invocation of the circuit. A $t$-limited adversary can probe up to $t$ wires in the circuit, and has unlimited computational power. Given a stateless circuit $C$, we must transform it into a new circuit $C'$ that can resist such an adversary. However, this is only possible if the inputs and outputs of the new circuit $C'$ are hidden since an input of $C$ might contain some secret-key bits and by probing these bits the adversary can obtain information about the secret key. Therefore, we allow the use of a randomized *input encoder* $I$ and *output decoder* $O$, whose wires can not be probed by the adversary. Both $I$ and $O$ should be independent from the circuit $C$ being transformed.

**Definition 3.1.** *Let $T$ be an efficiently computable, deterministic function mapping a stateless circuit $C$ to a stateless circuit $C'$, and let $I$, $O$ be input and output decoder, respectively. $(T, I, O)$ is said to be a $t$-private stateless transformer if it satisfies soundness and privacy, defined as follows:*

- *Soundness: $C$ and $O \circ C' \circ I$ have identical input-output functionality.*
- *Privacy: the values of any $t$ wires of $C'$ can be efficiently simulated without access to any wire of $C'$.*

In our conversion algorithms we will often work with $k$-bit variables (for some fixed parameter $k$) instead of single bits; in this case probing one such variable will automatically reveal its $k$-bit value instead of a single bit. Clearly, this can only make the adversary stronger.

The ISW framework also includes definitions for stateful circuits, i.e. circuits with memory gates. As shown in [ISW03], achieving privacy for stateful circuits is easy once privacy has been achieved in the stateless model. Thus, we focus on the stateless model in our work. We recall the main theorem from [ISW03] below.

**Theorem 3.1** (Ishai, Sahai, Wagner [ISW03])**.** *There exists a perfectly $t$-private stateless transformer $(T, I, O)$ such that $T$ maps any stateless circuit $C$ of size $|C|$ and depth $d$ to a randomized stateless circuit of size $\mathcal{O}(n^2 \cdot |C|)$ and depth $\mathcal{O}(d \log t)$, where $n = 2t + 1$.*

#### Privacy for Stateless Circuits.

For an arbitrary circuit $C$ the corresponding circuit $C'$ is constructed by maintaining the following invariant: for each wire in the circuit $C$, there are $n$ wires in $C'$, which add up to the value on the wire in $C$. Without loss of generality, any circuit $C$ can be represented using NOT and AND gates only. Thus, if we can transform these two gates, the whole circuit is transformable. It is easy to transform a NOT gate using the following simple relation: If $x = x_1 \oplus x_2 \oplus \cdots \oplus x_n$ then $\text{NOT}(x) = \text{NOT}(x_1) \oplus x_2 \oplus \cdots \oplus x_n$. To transform AND gates, the authors present an elegant solution, which is shown in Algorithm 13.

**Algorithm 13** ISWSecAnd

```
Input: $(x_i)$ and $(y_i)$ for $1 \leq i \leq n$
Output: $(z_i)$ for $1 \leq i \leq n$, with $\bigoplus_{i=1}^{n} z_i = \bigoplus_{i=1}^{n} x_i \wedge \bigoplus_{i=1}^{n} y_i$
 1: for $i = 1$ to $n$ do
 2:     for $j = i + 1$ to $n$ do
 3:         $r_{i,j} \leftarrow \mathsf{rand}(1)$
 4:         $r_{j,i} \leftarrow (r_{i,j} \oplus (x_i \wedge y_j)) \oplus (x_j \wedge y_i)$
 5:     end for
 6: end for
 7: for $i = 1$ to $n$ do
 8:     $z_i = x_i \wedge y_i$
 9:     for $j = 1$ to $n$ do
10:         if $i \neq j$ then
11:             $z_i \leftarrow z_i \oplus r_{i,j}$
12:         end if
13:     end for
14: end for
```

## 3.4 Secure Addition on Boolean Shares

In this section, we describe algorithms that can be used to perform an addition (or a subtraction) on the Boolean shares directly, thereby eliminating the need to convert masks from one form to the other. Formally, given $n$ Boolean shares of $x = x_1 \oplus \cdots \oplus x_n$ and $y = y_1 \oplus \cdots \oplus y_n$, we need to compute $n$ Boolean shares of $z = z_1 \oplus \cdots \oplus z_n$ satisfying the relation $z = x + y$, i.e.

$$z_1 \oplus \cdots \oplus z_n = (x_1 \oplus \cdots \oplus x_n) + (y_1 \oplus \cdots \oplus y_n)$$

We propose two algorithms to solve this problem based on the ISW method.

### 3.4.1 First Variant

The first solution is obtained by transforming the $k$-bit addition circuit into a circuit of XOR and AND gates so that the the ISW technique can be applied directly [ISW03]. A modular addition of two $k$-bit variables $x$ and $y$ can be defined

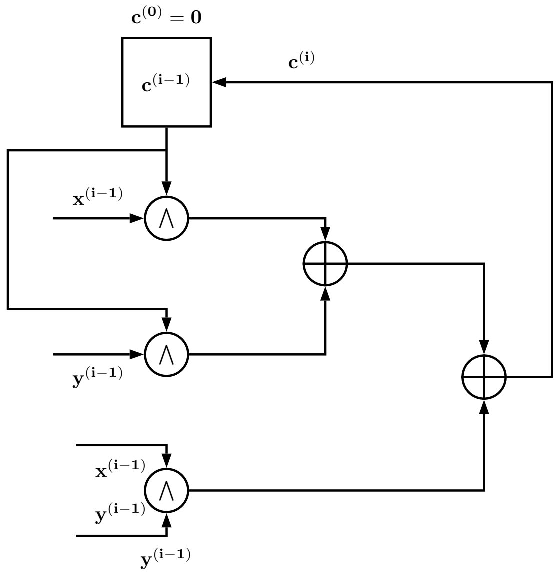


Figure 3.1: Recursive carry computation

recursively as $(x+y)^{(i)} = x^{(i)} \oplus y^{(i)} \oplus c^{(i)}$, where

$$\begin{cases} c^{(0)} = 0 \\ \forall i \geq 1, c^{(i)} = (x^{(i-1)} \wedge y^{(i-1)}) \oplus (x^{(i-1)} \wedge c^{(i-1)}) \oplus (c^{(i-1)} \wedge y^{(i-1)}) \end{cases} \tag{3.1}$$

Here, $x^{(i)}$ denotes the $i$-th bit of variable $x$, with $x^{(0)}$ being the least significant bit (refer to Figure 3.1). Since this recursion formula involves solely XOR and AND operations, we can simply use the ISW approach from [ISW03] to protect it against attacks of any order. The resulting algorithm is shown in Algorithm 14.

**Algorithm 14** SecAdd

```
Input: $(x_i)$ and $(y_i)$ for $1 \leq i \leq n$
Output: $(z_i)$ for $1 \leq i \leq n$, with $\bigoplus_{i=1}^{n} z_i = \bigoplus_{i=1}^{n} x_i + \bigoplus_{i=1}^{n} y_i$
1: $(c_i^{(0)})_{1\leq i\leq n} \leftarrow 0$    ▷ Initially carry is zero
2: for $j = 0$ to $k-2$ do    ▷ Compute carry bit by bit
3:     $(xy_i^{(j)})_{1\leq i\leq n} \leftarrow \mathsf{ISWSecAnd}((x_i^{(j)})_{1\leq i\leq n}, (y_i^{(j)})_{1\leq i\leq n})$    ▷ $x^{(j)} \wedge y^{(j)}$
4:     $(xc_i^{(j)})_{1\leq i\leq n} \leftarrow \mathsf{ISWSecAnd}((x_i^{(j)})_{1\leq i\leq n}, (c_i^{(j)})_{1\leq i\leq n})$    ▷ $x^{(j)} \wedge c^{(j)}$
5:     $(yc_i^{(j)})_{1\leq i\leq n} \leftarrow \mathsf{ISWSecAnd}((y_i^{(j)})_{1\leq i\leq n}, (c_i^{(j)})_{1\leq i\leq n})$    ▷ $y^{(j)} \wedge c^{(j)}$
6:     $(c_i^{(j+1)})_{1\leq i\leq n} \leftarrow (xy_i^{(j)})_{1\leq i\leq n} \oplus (xc_i^{(j)})_{1\leq i\leq n} \oplus (yc_i^{(j)})_{1\leq i\leq n}$
7: end for
8: $(z_i)_{1\leq i\leq n} \leftarrow (x_i)_{1\leq i\leq n} \oplus (y_i)_{1\leq i\leq n} \oplus (c_i)_{1\leq i\leq n}$    ▷ $z = x + y = x \oplus y \oplus c$
9: return $(z_i)_{1\leq i\leq n}$
```

Initially, there will be no carry; therefore, we set all $n$ shares of the carry to zero (Step 1). Next, we compute the carries for the remaining bits through the formula given in Equation (3.1). The loop runs from 0 to $k-2$ only, since the carry from the last bit does not need to be computed in a modular addition. In Step 8 we apply an XOR operation on the two inputs $x_i$, $y_i$ and the carry $c_i$ to obtain the $n$ shares corresponding to $x+y \bmod 2^k$. The algorithm ISWSecAnd has a time complexity of $\mathcal{O}(n^2)$ and, as a consequence, the full algorithm has a time complexity of $\mathcal{O}(n^2k)$. Algorithm 14 has to perform AND and XOR operations only. Due to the ISW scheme, we already know that such a circuit is protected from attacks of order $t$, where $n \geq 2t+1$. This proves the following theorem and shows the security of Algorithm 14 in the ISW model.

**Theorem 3.2.** *Let $(x_i)_{1\leq i\leq n}$ and $(y_i)_{1\leq i\leq n}$ be the input shares of Algorithm 14 and let $2t < n$. For any set of $t$ intermediate variables, there exists a subset $I \subset [1,n]$ of indices such that $|I| \leq n-1$, whereby the shares $x_{|I}$ and $y_{|I}$ can perfectly simulate those $t$ intermediate variables as well as the output shares $z_{|I}$.*

### 3.4.2 Second Variant

The second approach is based on the recursion from Goubin's theorem (Theorem 2.3), which uses the relation $x+y = x \oplus y \oplus u_{k-1}$, where $u_{k-1}$ is obtained from the following recursion formula:

$$\begin{cases} u_0 = 0 \\ \forall i \geq 0, u_{i+1} = 2[u_i \wedge (x \oplus y) \oplus (x \wedge y)] \end{cases}$$

Algorithm 15 presents the solution based on Goubin's formula to compute the addition. Here, the function ISWSecAnd is called with arguments of a size of $k$ bits instead of 1-bit arguments as in Algorithm 14. In this setting, the ISW scheme has to be adapted as follows: (i) all 1-bit variables defined over $\mathbb{F}_2$ are replaced by $k$-bit variables defined over $\mathbb{F}_{2^k}$; (ii) the 1-bit XOR operations are replaced by $k$-bit XOR operations; and (iii) the 1-bit AND operations are replaced by $k$-bit AND operations. This extension still preserves the security of the original scheme. Note that this method has been used before in the higher-order secure masking technique for AES proposed by Rivain and Prouff [RP10].[1]

The time complexity of Algorithm 15 is still $\mathcal{O}(n^2k)$. However, in practice, this algorithm will be faster for two reasons: (i) the number of calls to the function ISWSecAnd inside the loop is reduced from three to one, and (ii) all the operations are directly performed on the $k$-bit variables instead of single bits, thus there is no need to perform bit manipulations. Similar to Algorithm 14, it is easy to see that the security of Algorithm 15 follows from the original ISW scheme.

**Theorem 3.3.** *Let $(x_i)_{1\leq i\leq n}$ and $(y_i)_{1\leq i\leq n}$ be the input shares of Algorithm 15 and let $2t < n$. For any set of $t$ intermediate variables, there exists a subset $I \subset [1,n]$ of indices such that $|I| \leq n-1$, whereby the shares $x_{|I}$ and $y_{|I}$ can perfectly simulate those $t$ intermediate variables as well as the output shares $z_{|I}$.*

[1]In the Rivain-Prouff masking scheme, the AND operations over $\mathbb{F}_2$ were replaced with multiplications over $\mathbb{F}_{2^k}$ instead of AND operations over $\mathbb{F}_{2^k}$.

**Algorithm 15** SecAddGoubin

```
Input: $(x_i)$ and $(y_i)$ for $1 \le i \le n$
Output: $(z_i)$ for $1 \le i \le n$, with $\bigoplus_{i=1}^{n} z_i = \bigoplus_{i=1}^{n} x_i + \bigoplus_{i=1}^{n} y_i$
1: $(w_i)_{1\le i\le n} \leftarrow \mathsf{ISWSecAnd}((x_i)_{1\le i\le n}, (y_i)_{1\le i\le n})$    ▷ $\omega = x \wedge y$
2: $(u_i)_{1\le i\le n} \leftarrow 0$    ▷ Initialize shares of $u$ to zero
3: $(a_i)_{1\le i\le n} \leftarrow (x_i)_{1\le i\le n} \oplus (y_i)_{1\le i\le n}$    ▷ $a = x \oplus y$
4: for $j = 1$ to $k-1$ do
5:     $(ua_i)_{1\le i\le n} \leftarrow \mathsf{ISWSecAnd}\big((u_i)_{1\le i\le n}, (a_i)_{1\le i\le n}\big)$
6:     $(u_i)_{1\le i\le n} \leftarrow (ua_i)_{1\le i\le n} \oplus (w_i)_{1\le i\le n}$
7:     $(u_i)_{1\le i\le n} \leftarrow 2(u_i)_{1\le i\le n}$    ▷ $u \leftarrow 2(u \wedge a \oplus \omega)$
8: end for
9: $(z_i)_{1\le i\le n} \leftarrow (x_i)_{1\le i\le n} \oplus (y_i)_{1\le i\le n} \oplus (u_i)_{1\le i\le n}$    ▷ $z = x + y = x \oplus y \oplus u$
10: return $(z_i)_{1\le i\le n}$
```

## 3.5 Secure Arithmetic to Boolean Masking for Any Order

In this section, we describe two new algorithms for conversion from arithmetic to Boolean masking of any order. That is, given $n$ arithmetic shares with the property $x = A_1 + \cdots + A_n$, our algorithms output the corresponding Boolean shares satisfying $x = x_1 \oplus \cdots \oplus x_n$, secure against attacks of order $t$, where $2t \le n-1$. We describe in Section 3.6 the algorithm for secure conversion in the other direction, i.e. from Boolean to arithmetic masking.

We first present a straightforward algorithm with complexity $\mathcal{O}(n^3k)$, where $n$ and $k$ are the number of shares and the register size, respectively. Then, we give an improved algorithm with a complexity of $\mathcal{O}(n^2k)$. Internally, both algorithms use the secure addition function we described in Section 3.4. Though it is more efficient in practice to perform secure addition directly on Boolean shares (due to the overhead of converting between the masks twice), such conversion algorithms may still be useful, e.g. when the required number of conversions is lower than the required number of secure additions.[2]

### 3.5.1 A Simple Algorithm with Complexity $\mathcal{O}(n^3k)$

We first describe a simple approach for converting from arithmetic to Boolean masking with complexity $\mathcal{O}(n^3k)$. Assume that a sensitive variable $x$ is shared among $n$ arithmetic masks as follows:

$$x = A_1 + \cdots + A_n \tag{3.2}$$

We separately re-share each of the arithmetic shares $A_i$ $(1 \le i \le n)$ into $n$ random Boolean shares $x_{i,j}$ $(1 \le j \le n)$ so that $A_i = x_{i,1} \oplus \cdots \oplus x_{i,n}$. Hence, the sensitive

[2]For HMAC-SHA-1, it is more efficient to perform secure addition directly on the Boolean shares, as we will show later.

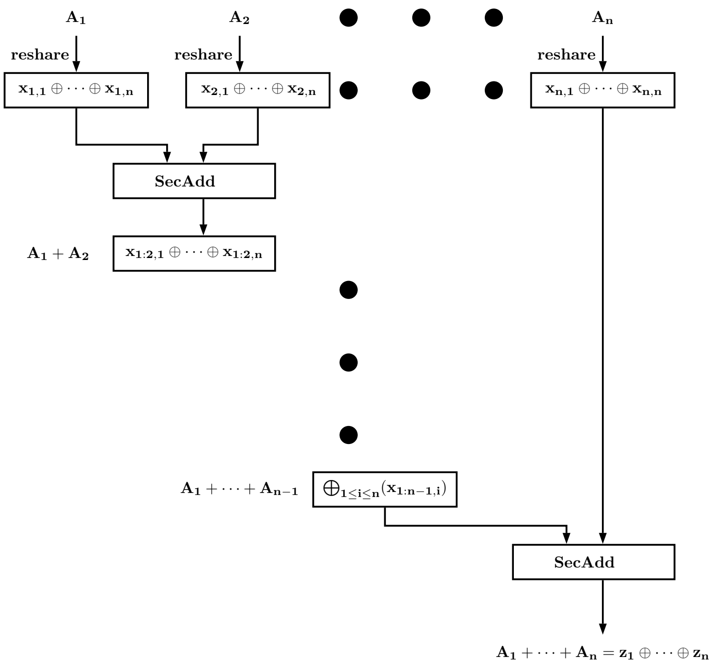

$\mathbf{A_1}$
$\mathbf{A_2}$
$\mathbf{A_n}$
reshare
reshare
reshare
$\mathbf{x_{1,1} \oplus \cdots \oplus x_{1,n}}$
$\mathbf{x_{2,1} \oplus \cdots \oplus x_{2,n}}$
$\mathbf{x_{n,1} \oplus \cdots \oplus x_{n,n}}$
SecAdd
$\mathbf{A_1 + A_2}$
$\mathbf{x_{1:2,1} \oplus \cdots \oplus x_{1:2,n}}$
$\mathbf{A_1 + \cdots + A_{n-1}}$
$\bigoplus_{1 \le i \le n}(\mathbf{x_{1:n-1,i}})$
SecAdd
$\mathbf{A_1 + \cdots + A_n = z_1 \oplus \cdots \oplus z_n}$

Figure 3.2: Arithmetic to Boolean conversion: Simple solution

variables $x$ is now given as:

$$x = (x_{1,1} \oplus \cdots \oplus x_{1,n}) + \cdots + (x_{n,1} \oplus \cdots \oplus x_{n,n}) \tag{3.3}$$

For each arithmetic share $A_i$ ($1 \le i \le n$), such re-sharing can be accomplished by generating $x_{i,j}$ independently at random for $2 \le j \le n$ and letting $x_{i,1} = A_i \oplus x_{i,2} \oplus \cdots \oplus x_{i,n}$. We then sequentially add the $A_i$'s using their $n$-Boolean shared representation $A_i = \bigoplus_{j=1}^{n} x_{i,j}$. For this, we use either the SecAdd or the SecAddGoubin algorithm from Section 3.4. Eventually, we get the final result $x$ in Boolean form as

$$x = z_1 \oplus \cdots \oplus z_n \tag{3.4}$$

This procedure is shown pictorially in Figure 3.2. Since each of the $n-1$ calls to SecAdd has a complexity of $\mathcal{O}(n^2 k)$, the overall complexity of the arithmetic to Boolean conversion is $\mathcal{O}(n^3 k)$.

**Theorem 3.4.** *Let $(A_i)_{1 \le i \le n}$ be the input shares of the previous algorithm and let $2t < n$. For any set of $t$ intermediate variables, there exists a subset $I \subset [1, n]$ of indices such that $|I| \le 2t < n$, whereby the shares $A_{|I}$ can perfectly simulate those $t$ intermediate variables as well as the output shares $z_{|I}$.*

*Proof.* We show how to simulate any set of $t$ probes, for $2t < n$. We firstly consider the initial re-sharing of the arithmetic shares $A_i$ ($1 \le i \le n$). At first, the set $I$ is empty. If there is a probe in the re-sharing of $A_i$, we add the index $i$ to $I$. Then, we consider the second part of the algorithm, starting from Equation (3.3) to the final result given by Equation (3.4). This second part is essentially an iteration of a circuit obtained through the ISW transform. Therefore, by applying the ISW methodology, we can simply continue with the construction of the subset $I$, so that any probe in this second part, and any of the output shares $z_{|I}$, can be perfectly simulated by knowing the inputs $x_{i,j}$ for $j \in I$ and for all $1 \leq i \leq n$; moreover, we know from the ISW methodology that $|I| \leq 2t < n$.

For any $i \notin I$, since the re-sharing of $A_i$ is not probed, we can perfectly simulate the $x_{i,j}$ for $j \in I$ without knowing $A_i$. Namely, since $|I| \leq 2t < n$, the $x_{i,j}$ for $j \in I$ form a proper subset of $n$ shares, and we can perfectly simulate such a subset without knowing $A_i$ by generating the values independently and uniformly at random. For $i \in I$, we can simulate the $x_{i,j}$ in the same way as in the "real" circuit because we know the input $A_i$. Therefore, as required, we can perfectly simulate the $x_{i,j}$ for $j \in I$ and all $1 \leq i \leq n$.

In summary, the $t$ probes as well as the output shares $z_{|I}$ can be perfectly simulated from the knowledge of the input shares $A_{|I}$, where $|I| \leq 2t < n$. □

It is easy to observe that one can improve the complexity of this algorithm by using fewer shares at the beginning. In particular, Equation (3.3) contains a total of $n^2$ shares, while only $n$ are necessary. Therefore, at the beginning, we use only two shares for every $A_i$ instead of $n$ shares. Then, we build a tree where at each layer the number of additive terms is divided by two, while the number of Boolean shares within an additive term is doubled. In this way, the overall number of shares remains $n$ or $2n$ at each level, and so the complexity becomes $\mathcal{O}(n^2k)$ instead of $\mathcal{O}(n^3k)$. We provide a complete description below.

### 3.5.2 Our New Arithmetic to Boolean Conversion Algorithm

In this section, we describe our new algorithm for converting from arithmetic to Boolean masking with a complexity of $\mathcal{O}(n^2k)$. Our algorithm is best described recursively. Assume that we already found an algorithm $\mathcal{A}_{n/2}$ for converting a set of $n/2$ arithmetic shares $A_i$ into $n/2$ Boolean shares $x_i$ such that

$$A_1 + \cdots + A_{n/2} = x_1 \oplus \cdots \oplus x_{n/2}.$$

Now, given as input a variable $x$ represented with $n$ arithmetic shares $A_i$:

$$x = A_1 + \cdots + A_n$$

we can first apply algorithm $\mathcal{A}_{n/2}$ separately on the two halves to get

$$\begin{aligned} x &= (A_1 + \cdots + A_{n/2}) + (A_{n/2+1} + \cdots + A_n) \\ &= (x_1 \oplus \cdots \oplus x_{n/2}) + (y_1 \oplus \cdots \oplus y_{n/2}) \end{aligned}$$

We now apply a simple expansion step, in which the $n/2$ shares $x_i$ and $y_i$ are each expanded to $n$ shares. This can be done by randomly splitting every share $x_i$ into $x_i = x'_{2i-1} \oplus x'_{2i}$ and similarly for $y_i = y'_{2i-1} \oplus y'_{2i}$. We obtain:

$$x = (x'_1 \oplus \cdots \oplus x'_n) + (y'_1 \oplus \cdots \oplus y'_n)$$

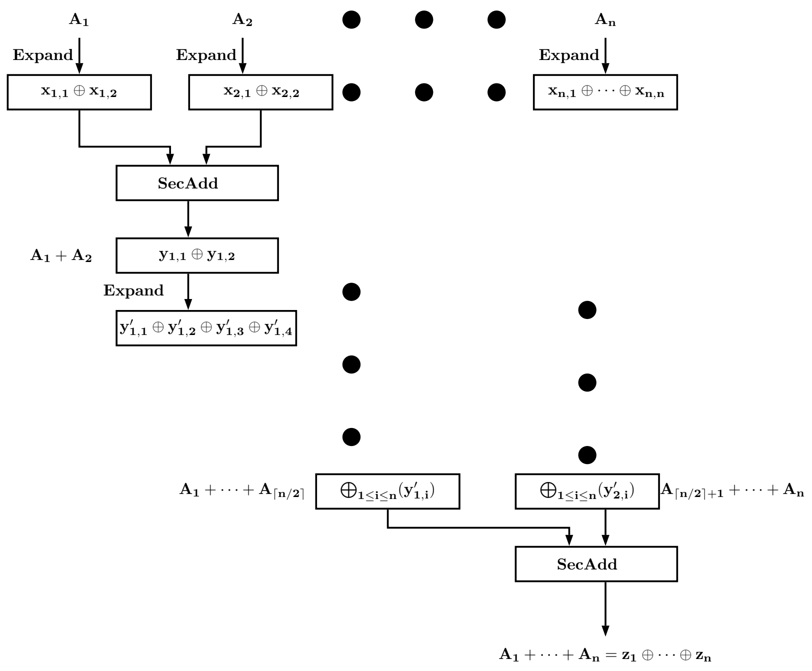

$A_1$
$A_2$
$A_n$
Expand
$x_{1,1} \oplus x_{1,2}$
$x_{2,1} \oplus x_{2,2}$
$x_{n,1} \oplus \cdots \oplus x_{n,n}$
SecAdd
$A_1 + A_2$
$y_{1,1} \oplus y_{1,2}$
$y'_{1,1} \oplus y'_{1,2} \oplus y'_{1,3} \oplus y'_{1,4}$
$A_1 + \cdots + A_{\lceil n/2 \rceil}$
$\bigoplus_{1 \le i \le n}(y'_{1,i})$
$\bigoplus_{1 \le i \le n}(y'_{2,i})$
$A_{\lceil n/2 \rceil + 1} + \cdots + A_n$
$A_1 + \cdots + A_n = z_1 \oplus \cdots \oplus z_n$

Figure 3.3: Arithmetic to Boolean conversion: Efficient solution

Then, we apply the $n$-Boolean addition circuit SecAdd or SecAddGoubin from Section 3.4 to obtain $x$ represented with $n$ Boolean shares $x = z_1 \oplus \cdots \oplus z_n$ as required (refer to Figure 3.3).

We now show that the algorithm has a complexity of $\mathcal{O}(n^2 k)$. For the sake of simplicity, we assume that $n$ is a power of two. Let $T_i$ be the execution time of $\mathcal{A}_i$, which takes $i$ arithmetic shares as input. We proceed by induction, based on the assumption that $T_i \le c \cdot i^2$ for all $i \le n/2$ and some constant $c$. When running algorithm $\mathcal{A}_n$ with $n$ shares, one first applies $\mathcal{A}_{n/2}$ on both halves, and then executes the expansion step (with $3n$ steps). Finally, the SecAdd algorithm is performed, which gives:

$$T_n \le 2T_{n/2} + 3n + c' \cdot n^2 \le 2c \cdot (n/2)^2 + 3n + c' \cdot n^2$$

for some constant $c'$, such that the execution time of SecAdd with $n$ shares is $\le c' \cdot n^2$. We get:

$$T_n \le (c/2 + 3 + c') \cdot n^2$$

Hence, it suffices to fix the constant $c$ so that $3 + c' \le c/2$ to get $T_n \le c \cdot n^2$ as required to prove the result. A formal description of our new conversion method can be found in Algorithm 16, which, in turn, uses the expansion step specified in Algorithm 17. The following theorem confirms that Algorithm 16 is secure in the ISW framework.

**Theorem 3.5.** *Let $(A_i)_{1 \le i \le n}$ be the input shares of Algorithm 16. For any set of $t$ intermediate variables and any $k$ output shares, there exists a subset $I \subset [1, n]$ of*

**Algorithm 16** ConvertA→B

```
Input: $(A_i)$ for $1 \le i \le n$
Output: $(z_i)$ for $1 \le i \le n$, with $\bigoplus_{i=1}^{n} z_i = \sum_{i=1}^{n} A_i$
1: If $n = 1$ then return $A_1$
2: $(x_i)_{1 \le i \le n/2} \leftarrow \mathsf{ConvertA{\to}B}\left((A_i)_{1 \le i \le n/2}\right)$
3: $(x'_i)_{1 \le i \le n} \leftarrow \mathsf{Expand}\left((x_i)_{1 \le i \le n/2}\right)$    ▷ $\bigoplus_{i=1}^{n} x'_i = \bigoplus_{i=1}^{n/2} x_i = \sum_{i=1}^{n/2} A_i$
4: $(y_i)_{1 \le i \le n/2} \leftarrow \mathsf{ConvertA{\to}B}\left((A_i)_{n/2+1 \le i \le n}\right)$
5: $(y'_i)_{1 \le i \le n} \leftarrow \mathsf{Expand}\left((y_i)_{1 \le i \le n/2}\right)$    ▷ $\bigoplus_{i=1}^{n} y'_i = \bigoplus_{i=1}^{n/2} y_i = \sum_{i=n/2+1}^{n} A_i$
6: $(z_i)_{1 \le i \le n} \leftarrow \mathsf{SecAdd}\left((x'_i)_{1 \le i \le n},\ (y'_i)_{1 \le i \le n}\right)$
7: return $(z_i)_{1 \le i \le n}$    ▷ $\bigoplus_{i=1}^{n} z_i = \bigoplus_{i=1}^{n} x'_i + \bigoplus_{i=1}^{n} y'_i = \sum_{i=1}^{n} A_i$
```

**Algorithm 17** Expand

```
Input: $x_i$ for $1 \le i \le n$
Output: $y_i$ for $1 \le i \le 2n$ with $\bigoplus_{i=1}^{2n} y_i = \bigoplus_{i=1}^{n} x_i$
1: $(r_i)_{1 \le i \le n} \leftarrow \mathsf{Rand}(k)$
2: $(y_{2i})_{1 \le i \le n} \leftarrow (x_i \oplus r_i)_{1 \le i \le n}$
3: $(y_{2i+1})_{1 \le i \le n} \leftarrow (r_i)_{1 \le i \le n}$
4: return $(y_i)_{1 \le i \le 2n}$
```

*indices such that $|I| \le k + 2t$, where the shares $A_{|I}$ can perfectly simulate those $t$ intermediate variables as well as the output shares $x_{|I}$.*

*Proof.* We first prove the following property of the Expand method.

**Lemma 3.1.** *In Algorithm 17, a set of $k$ outputs ($k \le 2n$) and $t$ probes ($t \le n$) can be perfectly simulated using at most $\lfloor k/2 \rfloor + t$ inputs.*

*Proof of Lemma 3.1.* We proceed by induction. When $n = 1$, the algorithm gets only $x$ as input and outputs $(x \oplus r, r)$ for a uniformly random $r$. Now, we have to distinguish between the following two cases: there is no probe ($t = 0$), and there is at least one probe ($t \ge 1$).

In the latter case, i.e. there is at least one probe (for $x$, or $r$, or $x \oplus r$), then $t \ge 1$ and the probe can be perfectly simulated by using the input $x$ and generating $r$ uniformly at random. This will also perfectly simulate both outputs. As a consequence, for $t = 1$ and any $k$ with $0 \le k \le 2$, we can perfectly simulate the $t$ probes and the $k$ outputs using at most $1 \le \lfloor k/2 \rfloor + t$ inputs.

We now assume that there are no probes ($t = 0$). If no output needs to be simulated (i.e. $k = 0$), then knowledge of the input $x$ is not required. If only a single output must be simulated ($k = 1$), where either $y_1 = x \oplus r$ or $y_2 = r$ has to be simulated, such output can be perfectly simulated by generating a random number uniformly, without knowing $x$. Finally, if $k = 2$, then one input is required. Therefore, for any $k$ with $0 \le k \le 2$, the number of required inputs is always at most $\lfloor k/2 \rfloor + t$.

For $n > 1$, let us consider the $i$-th sub-circuit and denote the number of outputs to be simulated by $k_i$ and the number of probes by $t_i$ for $1 \leq i \leq n$. Based on the above arguments, the total number of inputs needed for the simulation is then at most

$$\sum_{i=1}^{n} \lfloor k_i/2 \rfloor + t_i \leq \lfloor k/2 \rfloor + t,$$

which finally proves the Lemma. □

The proof of Theorem 3.5 is obtained via induction on the number of shares $n$. We assume that the result holds for $n/2$ and prove that it holds for $n$. We distinguish among 5 sets of probes:

- The $t_A$ probes for the Secure Addition subroutine (Line 6 of Algorithm 16).
- The $t_{EL}$ and $t_{ER}$ probes for the left and right Expand circuit, respectively (lines 3 and 5 of Algorithm 16).
- The $t_{CL}$ and $t_{CR}$ probes for the left and right Arithmetic to Boolean conversion circuit, respectively (lines 2 and 4 of Algorithm 16).

From the security proof of the ISWSecAnd algorithm given in [ISW03], we know that a set of $k$ outputs and $t_A$ probes can be simulated using a subset of $k + 2t_A$ inputs in each of the two input shares $x'_i$ and $y'_i$. Therefore, the property also holds for the SecAdd algorithm.

According to Lemma 3.1, a set of $k + 2t_A$ outputs and $t_{EL}$ (resp. $t_{ER}$) probes can be simulated using at most $\lfloor (k + 2t_A)/2 \rfloor + t_{EL} = \lfloor k/2 \rfloor + t_A + t_{EL}$ inputs (resp. $\lfloor k/2 \rfloor + t_A + t_{ER}$ inputs). Since the result is assumed to hold for $n/2$, the $\lfloor k/2 \rfloor + t_A + t_{EL}$ outputs and the $t_{CL}$ probes of the left conversion can be simulated using at most $\lfloor k/2 \rfloor + t_A + t_{EL} + 2t_{CL}$ inputs. An upper bound of the number of inputs for the right conversion can be derived in the same way. As a consequence, the total number of required inputs is at most $k + 2t$ according to the following equation

$$\begin{aligned} |I| &\leq \lfloor k/2 \rfloor + t_A + t_{EL} + 2t_{CL} + \lfloor k/2 \rfloor + t_A + t_{ER} + 2t_{CR} \\ &\leq k + 2(t_A + t_{EL} + t_{ER} + t_{CL} + t_{CR}) \\ &\leq k + 2t, \end{aligned}$$

which proves Theorem 3.5. □

## 3.6 From Boolean to Arithmetic Masking of Any Order

We now present a new algorithm for converting in the other direction, i.e. from Boolean to arithmetic masking, again with a complexity of $\mathcal{O}(n^2 k)$. Algorithm 18 specifies our arithmetic-to-Boolean conversion in detail.

**Algorithm 18** ConvertB→A

```
Input: $(x_i)$ for $1 \le i \le n$
Output: $(A_i)$ for $1 \le i \le n$, with $\sum_{i=1}^{n} A_i = \bigoplus_{i=1}^{n} x_i$
1: $(A_i)_{1 \le i \le n-1} \leftarrow \mathsf{Rand}(k)$
2: $(A'_i)_{1 \le i \le n-1} \leftarrow (-A_i)_{1 \le i \le n-1},\ A'_n \leftarrow 0$
3: $(y_i)_{1 \le i \le n} \leftarrow \mathsf{ConvertA{\to}B}\big((A'_i)_{1 \le i \le n}\big)$    ▷ $\bigoplus_{i=1}^{n} y_i = \sum_{i=1}^{n} A'_i = -\sum_{i=1}^{n-1} A_i$
4: $(z_i)_{1 \le i \le n} \leftarrow \mathsf{SecAdd}\big((x_i)_{1 \le i \le n}, (y_i)_{1 \le i \le n}\big)$    ▷ $\bigoplus_{i=1}^{n} z_i = \bigoplus_{i=1}^{n} x_i + \bigoplus_{i=1}^{n} y_i$
5: $A_n \leftarrow \mathsf{FullXor}\big((z_i)_{1 \le i \le n}\big)$    ▷ $A_n = \bigoplus_{i=1}^{n} z_i = \bigoplus_{i=1}^{n} x_i - \sum_{i=1}^{n-1} A_i$
6: return $(A_i)_{1 \le i \le n}$.    ▷ $\sum_{i=1}^{n} A_i = \bigoplus_{i=1}^{n} x_i$
```

We use the same randomized XOR method as in [Cor14] to compute $A_n \leftarrow \bigoplus_{i=1}^{n} z_i$; we recall this method in Algorithm 19. The randomized XOR method, in turn, uses Algorithm 20 (which was first proposed by Rivain and Prouff [RP10]) to refresh the masks.

**Algorithm 19** FullXor

```
Input: $y_1, \dots, y_n$
Output: $y$ such that $y = y_1 \oplus \cdots \oplus y_n$
1: for $i = 1$ to $n$ do $(y_1, \dots, y_n) \leftarrow \mathsf{RefreshMasks}(y_1, \dots, y_n)$
2: return $y_1 \oplus \cdots \oplus y_n$
```

**Algorithm 20** RefreshMasks

```
Input: $z_1, \dots, z_n$ such that $z = z_1 \oplus \cdots \oplus z_n$
Output: $z_1, \dots, z_n$ such that $z = z_1 \oplus \cdots \oplus z_n$
1: for $j = 2$ to $n$ do
2:     $tmp \leftarrow \mathsf{Rand}(k)$
3:     $z_1 \leftarrow z_1 \oplus tmp$
4:     $z_j \leftarrow z_j \oplus tmp$
5: end for
6: return $z_1, \dots, z_n$
```

The following theorem proves the security of Algorithm 18 in the ISW model.

**Theorem 3.6.** *Let $(x_i)_{1 \le i \le n}$ be the input shares of Algorithm 18. For any set of $t$ intermediate variables with $2t < n$, there exists a subset $I \subset [1, n]$ of indices such that $|I| \le 2t$, whereby the shares $x_{|I}$ can perfectly simulate those $t$ intermediate variables as well as the output shares $A_{|I}$.*

We recall the following Lemma from [Cor14] (with $|I| \le t$ instead of $|I| \le 2t$) and its proof.

**Lemma 3.2.** *Let $(y_i)_{1\leq i\leq n}$ be the input shares of the* FullXor *algorithm. For any set of $t$ intermediate variables, there exists a subset $I \subset [1, n]$ of indices such that $|I| \leq t$ and the distribution of those $t$ variables can be perfectly simulated from $y_{|I}$ and $y = y_1 \oplus \cdots \oplus y_n$.*

*Proof of Lemma 3.2.* We first consider the series of $n$ RefreshMasks. If any variable $y_j$ is probed inside any of the RefreshMasks, we add $j$ to $I$.

Moreover since $t < n$, there must be at least one RefreshMasks that is not probed at all; let $i^*$ be the index of this RefreshMasks. Since we know $y = y_1 \oplus \cdots \oplus y_n$, we can perfectly simulate *all* the shares $(y_i)_{1\leq i\leq n}$ after this $i^*$-th RefreshMasks. Therefore we can perfectly simulate all $y_i$'s until the last RefreshMasks, and all intermediate variables for computing $y = y_1 \oplus \cdots \oplus y_n$.

In summary before the $i^*$ RefreshMasks, with the knowledge of the input shares $y_{|I}$, we can perfectly simulate all intermediate variables $y_j$ for $j \in I$, and after the $i^*$ RefreshMasks we can perfectly simulate all intermediate variables. Finally the *tmp* variables are simulated as in the real circuit. This proves Lemma 3.2. □

From Lemma 3.2, the set of $t_1$ probes in the FullXor circuit computing $A_n = \bigoplus_{i=1}^{n} z_i$ can be simulated from $A_n$ and at most $t_1$ inputs $z_i$. From the previous lemmas, those $t_1$ inputs $z_i$ and the $t_2$ probes in the remaining circuit can be perfectly simulated using $x_{|I}$, for $I \subset [1, n]$, where $|I| \leq t_1 + 2t_2$. If $t_1 > 0$ we add $n$ to $I$; we still have $|I| \leq 2t$ where $t = t_1 + t_2$.

It remains to show how we can simulate $A_n$, as this is required for the simulation in Lemma 3.2 if $t_1 > 0$, or if $t_1 = 0$ and $n \in I$, since we must simulate all outputs $A_{|I}$. We select an arbitrary $i_0 \notin I$ such that $i_0 \neq n$; this is possible since in both cases we have $n \in I$ and $|I| \leq 2t < n$. We have:

$$A_n = \left( x - \sum_{\substack{i=1\\ i\neq i_0}}^{n-1} A_i \right) - A_{i_0}$$

Since $i_0 \notin I$ the variable $A_{i_0}$ does not enter in any computation of the simulation. Since in the real circuit $A_{i_0}$ is generated uniformly at random, we can simulate $A_n$ by generating a uniform random value. This proves Theorem 3.6.

## 3.7 Implementation Results

We have implemented all the solutions proposed in this chapter on a 32-bit AVR microcontroller for security level $t = 2, 3$. We then applied all these techniques to HMAC-SHA-1 and compared the running time with respect to an unmasked implementation. Table 3.1 gives the running time of the addition and conversion algorithms along with the number of calls to the rand function for security level $t = 2, 3$. As expected, the addition algorithms using Goubin's theorem (i.e. the second variant presented in Section 3.4.2) outperform the first variant (given in Section 3.4.1) which can be observed in Figure 3.4. Therefore, we applied the second variant to implement the secure conversion algorithms.

Table 3.1: Execution times of all algorithms (in thousands of clock cycles) for $t = 2, 3$ and the number of calls to the rand function

| Algorithm | Time | rand |
|---|---|---|
| second-order addition | | |
| Algorithm 14 | 87 | 1240 |
| Algorithm 15 | 26 | 320 |
| second-order conversion | | |
| Algorithm 16 | 54 | 484 |
| Algorithm 18 | 81 | 822 |
| third-order addition | | |
| Algorithm 14 | 156 | 2604 |
| Algorithm 15 | 46 | 672 |
| third-order conversion | | |
| Algorithm 16 | 121 | 1288 |
| Algorithm 18 | 162 | 1997 |

**HMAC-SHA-1.**

The hash function SHA-1 operates on blocks of 512 bits and produces a 160-bit message digest. Each message block is divided into 16 words of 32-bits each, which are extended to produce 64 further words (i.e. the total number of words is 80). The main loop contains 80 iterations corresponding to each of these 80 words. In order to protect HMAC-SHA-1 against side-channel attacks, we follow two different approaches, which are summarized below.

In the first approach, we use Boolean masking and perform secure addition on Boolean shares directly whenever required. Every iteration of the main loop requires four 32-bit additions, which amounts in a total of 320 additions for 80 iterations. Moreover, five additions have to be performed at the end to update the state. So, in total, 325 secure additions need to be carried out per message block.

In the other approach, we use Boolean masking and convert it to arithmetic masking wherever necessary. In this case, we need four Boolean to arithmetic conversions and one arithmetic to Boolean conversion per iteration, yielding a total of 400 conversions for 80 iterations. Additionally, we need 10 conversions to update the result, i.e. a total of 410 conversions per block are required. The execution times of both approaches are summarized in Table 3.2. For both $t = 2$ and $t = 3$, the implementation using SecAddGoubin algorithm produces the best results which can be seen in Figure 3.5.

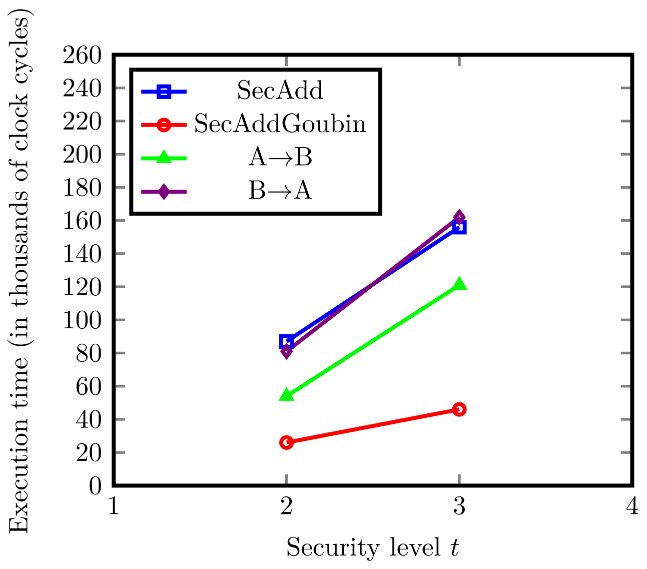


Figure 3.4: Comparison of execution times of all algorithms (in thousands of clock cycles) for $t = 2, 3$

| Algorithm | Time | Penalty |
|---|---|---|
| HMAC-SHA-1 | 104 | 1 |
| second-order addition | | |
| Algorithm 14 | 57172 | 549 |
| Algorithm 15 | 17847 | 171 |
| second-order conversion | | |
| Algorithm 16, 18 | 62669 | 602 |
| third-order addition | | |
| Algorithm 14 | 106292 | 987 |
| Algorithm 15 | 31195 | 299 |
| third-order conversion | | |
| Algorithm 16, 18 | 127348 | 1224 |

Table 3.2: Execution times of second and third-order secure masking (in thousands of clock cycles) and performance penalty compared to an unmasked implementation of HMAC-SHA-1

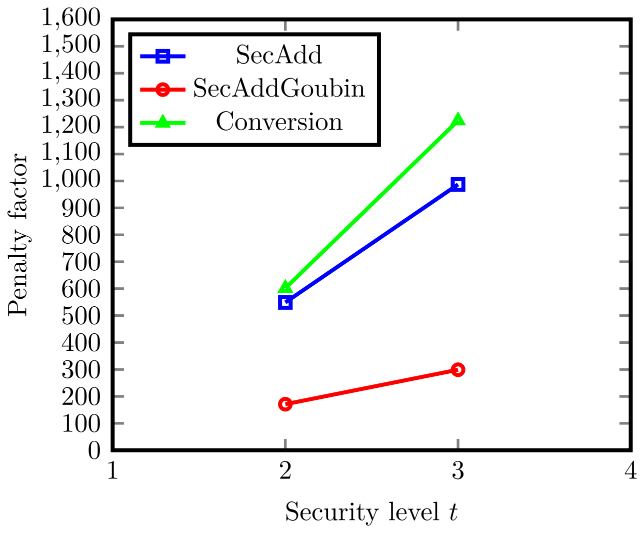


Figure 3.5: Comparison of performance penalty compared to an unmasked implementation of HMAC-SHA-1

# Chapter 4

# Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity

Goubin's algorithm for converting from arithmetic to Boolean masking requires $\mathcal{O}(k)$ operations where $k$ is the addition bit size. In this chapter we describe improved algorithm with time complexity $\mathcal{O}(\log k)$ only. Our new algorithm is based on the Kogge-Stone carry look-ahead adder, which computes the carry signal in $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$ for the classical ripple carry adder. We also describe an algorithm for performing arithmetic addition modulo $2^k$ directly on Boolean shares, with the same complexity $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$. We prove the security of our new algorithm against first-order attacks. Our algorithm performs well in practice, as for $k = 64$ we obtain a 23% improvement compared to Goubin's algorithm. In Chapter 3 we gave the first solutions to perform conversion between Boolean and arithmetic masking of any order. These algorithms have time complexity in $\mathcal{O}(n^2 \cdot k)$ for $n$ shares of $k$ bits each. In this chapter we give new algorithms with complexity $\mathcal{O}(n^2 \cdot \log k)$ instead of $\mathcal{O}(n^2 \cdot k)$ for $n$ shares, again using Kogge-Stone carry look-ahead adder. This improves the performance of the original algorithms by 70%. This is a joint work with Jean-Sébastien Coron, Johann Großschädl and Mehdi Tibouchi. A part of this work will appear in the proceedings of FSE, 2015 [CGVT15].

**Contents**

## 4.1 A New Recursive Formula Based on Kogge-Stone Adder

Our new conversion algorithm is based on the Kogge-Stone adder [KS73], a carry look-ahead adder that generates the carry signal in $\mathcal{O}(\log k)$ time, when addition is performed modulo $2^k$. In this section we first recall the classical ripple-carry adder, which generates the carry signal in $\mathcal{O}(k)$ time, and we show how Goubin's recursion formula (2.1) can be derived from it. The derivation of our new recursion formula from the Kogge-Stone adder will proceed similarly.

### 4.1.1 The Ripple-Carry Adder and Goubin's Recursion Formula

We first recall the classical ripple-carry adder. Given three bits $x$, $y$ and $c$, the carry $c'$ for $x+y+c$ can be computed as $c' = (x \wedge y) \oplus (x \wedge c) \oplus (y \wedge c)$. Therefore, the modular addition of two $k$-bit variables $x$ and $y$ can be defined recursively as follows:

$$(x+y)^{(i)} = x^{(i)} \oplus y^{(i)} \oplus c^{(i)} \tag{4.1}$$

for $0 \leq i < k$, where

$$\begin{cases} c^{(0)} = 0 \\ \forall i \geq 1,\ c^{(i)} = (x^{(i-1)} \wedge y^{(i-1)}) \oplus (x^{(i-1)} \wedge c^{(i-1)}) \oplus (c^{(i-1)} \wedge y^{(i-1)}) \end{cases} \tag{4.2}$$

where $x^{(i)}$ represents the $i^{\text{th}}$ bit of the variable $x$, with $x^{(0)}$ being the least significant bit.

In the following, we show how recursion (4.2) can be computed directly with $k$-bit values instead of bits, which enables us to recover Goubin's recursion (2.1). For this, we define the sequences $x_j$, $y_j$ and $v_j$ whose $j+1$ least significant bits are the same as $x$, $y$ and $c$ respectively:

$$x_j = \bigoplus_{i=0}^{j} 2^i x^{(i)}, \quad y_j = \bigoplus_{i=0}^{j} 2^i y^{(i)}, \quad v_j = \bigoplus_{i=0}^{j} 2^i c^{(i)} \tag{4.3}$$

for $0 \leq j \leq k-1$. Since $c^{(0)} = 0$ we can actually start the summation for $v_j$ at $i = 1$; we get from (4.2):

$$\begin{aligned}
v_{j+1} &= \bigoplus_{i=1}^{j+1} 2^i c^{(i)} \\
v_{j+1} &= \bigoplus_{i=1}^{j+1} 2^i \left( (x^{(i-1)} \wedge y^{(i-1)}) \oplus (x^{(i-1)} \wedge c^{(i-1)}) \oplus (c^{(i-1)} \wedge y^{(i-1)}) \right) \\
v_{j+1} &= 2 \bigoplus_{i=0}^{j} 2^i \left( (x^{(i)} \wedge y^{(i)}) \oplus (x^{(i)} \wedge c^{(i)}) \oplus (c^{(i)} \wedge y^{(i)}) \right) \\
v_{j+1} &= 2\big( (x_j \wedge y_j) \oplus (x_j \wedge v_j) \oplus (y_j \wedge v_j) \big)
\end{aligned}$$

which gives the recursive equation:

$$\begin{cases} v_0 = 0 \\ \forall j \geq 0,\ v_{j+1} = 2\left(v_j \wedge (x_j \oplus y_j) \oplus (x_j \wedge y_j)\right) \end{cases} \tag{4.4}$$

Therefore we have obtained a recursion similar to (4.2), but with $k$-bit values instead of single bits. Note that from the definition of $v_j$ in (4.3) the variables $v_j$ and $v_{j+1}$ have the same least significant bits from bit 0 to bit $j$, which is not immediately obvious when considering only recursion (4.4). Combining (4.1) and (4.3) we obtain $x_j + y_j = x_j \oplus y_j \oplus v_j$ for all $0 \leq j \leq k-1$. For $k$-bit values $x$ and $y$, we have $x = x_{k-1}$ and $y = y_{k-1}$, which gives:

$$x + y = x \oplus y \oplus v_{k-1}$$

We now define the same recursion as (4.4), but with constant $x$, $y$ instead of $x_j$, $y_j$. That is, we let

$$\begin{cases} u_0 = 0 \\ \forall j \geq 0,\ u_{j+1} = 2\left(u_j \wedge (x \oplus y) \oplus (x \wedge y)\right) \end{cases} \tag{4.5}$$

which is exactly the same recursion as Goubin's recursion (2.1). It is easy to show inductively that the variables $u_j$ and $v_j$ have the same least significant bits, from bit 0 to bit $j$. Let us assume that this is true for $u_j$ and $v_j$. From recursions (4.4) and (4.5) we have that the least significant bits of $v_{j+1}$ and $u_{j+1}$ from bit 0 to bit $j+1$ only depend on the least significant bits from bit 0 to bit $j$ of $v_j$, $x_j$ and $y_j$, and of $u_j$, $x$ and $y$ respectively. Since these are the same, the induction is proved.

Eventually for $k$-bit registers we have $u_{k-1} = v_{k-1}$, which proves Goubin's recursion formula (2.1), namely:

$$x + y = x \oplus y \oplus u_{k-1}$$

As mentioned previously, this recursion formula requires $k-1$ iterations on $k$-bit registers. In the following, we describe an improved recursion based on the Kogge-Stone carry look-ahead adder, requiring only $\log_2 k$ iterations.

### 4.1.2 The Kogge-Stone Carry Look-Ahead Adder

In this section we first recall the general solution from [KS73] for first-order recurrence equations; the Kogge-Stone carry look-ahead adder is a direct application.

#### General first-order recurrence equation.

We consider the following recurrence equation:

$$\begin{cases} z_0 = b_0 \\ \forall i \geq 1,\ z_i = a_i z_{i-1} + b_i \end{cases} \tag{4.6}$$

We define the function $Q(m,n)$ for $m \geq n$:

$$Q(m,n) = \sum_{j=n}^{m} \left( \prod_{i=j+1}^{m} a_i \right) b_j \tag{4.7}$$

We have $Q(0,0) = b_0 = z_0$, $Q(1,0) = a_1 b_0 + b_1 = z_1$, and more generally:

$$\begin{aligned} Q(m,0) &= \sum_{j=0}^{m-1} \left( \prod_{i=j+1}^{m} a_i \right) b_j + b_m \\ &= a_m \sum_{j=0}^{m-1} \left( \prod_{i=j+1}^{m-1} a_i \right) b_j + b_m = a_m Q(m-1,0) + b_m \end{aligned}$$

Therefore the sequence $Q(m,0)$ satisfies the same recurrence as $z_m$, which implies $Q(m,0) = z_m$ for all $m \geq 0$. Moreover we have:

$$\begin{aligned} Q(2m-1,0) &= \sum_{j=0}^{2m-1} \left( \prod_{i=j+1}^{2m-1} a_i \right) b_j \\ &= \left( \prod_{j=m}^{2m-1} a_j \right) \sum_{j=0}^{m-1} \left( \prod_{i=j+1}^{m-1} a_i \right) b_j + \sum_{j=m}^{2m-1} \left( \prod_{i=j+1}^{2m-1} a_i \right) b_j \end{aligned}$$

which gives the recursive doubling equation:

$$Q(2m-1,0) = \left( \prod_{j=m}^{2m-1} a_j \right) Q(m-1,0) + Q(2m-1,m)$$

where each term $Q(m-1,0)$ and $Q(2m-1,m)$ contain only $m$ terms $a_i$ and $b_i$, instead of $2m$ in $Q(2m-1,0)$. Therefore the two terms can be computed in parallel. This is also the case for the product $\prod_{j=m}^{2m-1} a_j$ which can be computed with a product tree. Therefore by recursive splitting with $N$ processors, the sequence element $z_N$ can be computed in time $\mathcal{O}(\log_2 N)$, instead of $\mathcal{O}(N)$ with a single processor.

**The Kogge-Stone Carry Look-Ahead Adder.**

The Kogge-Stone carry look-ahead adder [KS73] is a direct application of the previous technique. Namely writing $c_i = c^{(i)}$, $a_i = x^{(i)} \oplus y^{(i)}$ and $b_i = x^{(i)} \wedge y^{(i)}$ for all $i \geq 0$, we obtain from (4.2) the recurrence relation for the carry signal $c_i$:

$$\begin{cases} c_0 = 0 \\ \forall i \geq 1,\ c_i = (a_{i-1} \wedge c_{i-1}) \oplus b_{i-1} \end{cases}$$

which is similar to (4.6), where $\wedge$ is the multiplication and $\oplus$ the addition. We can therefore compute the carry signal $c_i$ for $0 \leq i < k$ in time $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$.

More precisely, the Kogge-Stone carry look-ahead adder can be defined as follows. For all $0 \leq j < k$ one defines the sequence of bits:

$$P_{0,j} = x^{(j)} \oplus y^{(j)}, \quad G_{0,j} = x^{(j)} \wedge y^{(j)} \tag{4.8}$$

and the following recursive equations:

$$\begin{cases} P_{i,j} &= P_{i-1,j} \wedge P_{i-1,j-2^{i-1}} \\ G_{i,j} &= (P_{i-1,j} \wedge G_{i-1,j-2^{i-1}}) \oplus G_{i-1,j} \end{cases} \tag{4.9}$$

for $2^{i-1} \leq j < k$, and $P_{i,j} = P_{i-1,j}$ and $G_{i,j} = G_{i-1,j}$ for $0 \leq j < 2^{i-1}$. The following lemma shows that the carry signal $c_j$ can be computed from the sequence $G_{i,j}$.

**Lemma 4.1.** *We have $(x+y)^{(j)} = x^{(j)} \oplus y^{(j)} \oplus c_j$ for all $0 \leq j < k$ where the carry signal $c_j$ is computed as $c_0 = 0$, $c_1 = G_{0,0}$ and $c_{j+1} = G_{i,j}$ for $2^{i-1} \leq j < 2^i$.*

*Proof.* To compute the carry signal up to $c_{k-1}$, one must therefore compute the sequences $P_{i,j}$ and $G_{i,j}$ up to $i = \lceil \log_2(k-1) \rceil$. For completeness we provide the proof of Lemma 4.1 in below.

We consider again recursion (4.6):

$$\begin{cases} z_0 = b_0 \\ \forall i \geq 1,\ z_i = a_i z_{i-1} + b_i \end{cases}$$

The recursion for $c_i$ is similar when we denote the AND operation by a multiplication, and the XOR operation by an addition:

$$\begin{cases} c_0 = 0 \\ \forall i \geq 1,\ c_i = a_{i-1} c_{i-1} + b_{i-1} \end{cases}$$

Therefore we obtain $c_{i+1} = z_i$ for all $i \geq 0$. From the $Q(m,n)$ function given in (4.7) we define the sequences:

$$\begin{aligned} G_{i,j} &:= Q(j, \max(j - 2^i + 1, 0)) \\ P_{i,j} &:= \prod_{v=\max(j-2^i+1,0)}^{j} a_v \end{aligned}$$

We show that these sequences satisfy the same recurrence (4.9). From (4.7) we have the recurrence for $j \geq 2^{i-1}$:

$$\begin{aligned}
G_{i,j} &= \sum_{u=\max(j-2^i+1,0)}^{j} \left( \prod_{v=u+1}^{j} a_v \right) b_u \\
&= \sum_{u=\max(j-2^i+1,0)}^{j-2^{i-1}} \left( \prod_{v=u+1}^{j} a_v \right) b_u + \sum_{u=j-2^{i-1}+1}^{j} \left( \prod_{v=u+1}^{j} a_v \right) b_u \\
&= \left( \prod_{v=j-2^{i-1}+1}^{j} a_v \right) \sum_{u=\max(j-2^i+1,0)}^{j-2^{i-1}} \left( \prod_{v=u+1}^{j-2^{i-1}} a_v \right) b_u + Q(j, j-2^{i-1}+1) \\
&= P_{i-1,j} \cdot Q(j-2^{i-1}, \max(j-2^i+1,0)) + G_{i-1,j} \\
&= P_{i-1,j} \cdot G_{i-1,j-2^{i-1}} + G_{i-1,j}
\end{aligned}$$

We obtain a similar recurrence for $P_{i,j}$ when $j \geq 2^{i-1}$:

$$\begin{aligned}
P_{i,j} &= \prod_{v=\max(j-2^i+1,0)}^{j} a_v \\
&= \left( \prod_{v=\max(j-2^i+1,0)}^{j-2^{i-1}} a_v \right) \cdot \left( \prod_{v=j-2^{i-1}+1}^{j} a_v \right) = P_{i-1,j-2^{i-1}} \cdot P_{i-1,j}
\end{aligned}$$

In summary we obtain for $j \geq 2^{i-1}$ the relations:

$$\begin{cases} G_{i,j} = P_{i-1,j} \cdot G_{i-1,j-2^{i-1}} + G_{i-1,j} \\ P_{i,j} = P_{i-1,j} \cdot P_{i-1,j-2^{i-1}} \end{cases}$$

which are exactly the same as (4.9). Moreover for $0 \leq j < 2^{i-1}$, we have $G_{i,j} = Q(j,0) = G_{i-1,j}$ and $P_{i,j} = P_{i-1,j}$. Finally we have the same initial conditions $G_{0,j} = Q(j,j) = b_j = x^{(j)} \wedge y^{(j)}$ and $P_{0,j} = a_j = x^{(j)} \oplus y^{(j)}$. This proves that the sequence $G_{i,j}$ defined by (4.9) is such that:

$$G_{i,j} = Q(j, \max(j-2^i+1, 0))$$

This implies that we have $G_{0,0} = Q(0,0) = z_0$ and $G_{i,j} = Q(j,0) = z_j$ for all $2^{i-1} \leq j < 2^i$. Moreover as noted initially we have $c_{j+1} = z_j$ for all $j \geq 0$. Therefore the recurrence indeed computes the carry signal $c_j$, with $c_0 = 0$, $c_1 = G_{0,0}$ and $c_{j+1} = G_{i,j}$ for $2^{i-1} \leq j < 2^i$. This terminates the proof of Lemma 4.1. □

### 4.1.3 Our New Recursive Algorithm

We now derive a recursion formula with $k$-bit variables instead of single bits; we proceed as in Section 4.1.1, using the more efficient Kogge-Stone carry look-ahead algorithm, instead of the classical ripple-carry adder for Goubin's recursion. We prove the following theorem, analogous to Theorem 2.3, but with complexity $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$. Given a variable $x$, we denote by $x \ll \ell$ the variable $x$ left-shifted by $\ell$ bits, keeping only $k$ bits in total.

**Theorem 4.1.** *Let $x, y \in \{0,1\}^k$ and $n = \lceil \log_2(k-1) \rceil$. Define the sequence of $k$-bit variables $P_i$ and $G_i$, with $P_0 = x \oplus y$ and $G_0 = x \wedge y$, and*

$$\left\{ \begin{array}{lcl} P_i & = & P_{i-1} \wedge (P_{i-1} \ll 2^{i-1}) \\ G_i & = & (P_{i-1} \wedge (G_{i-1} \ll 2^{i-1})) \oplus G_{i-1} \end{array} \right. \tag{4.10}$$

*for $1 \le i \le n$. Then $x + y = x \oplus y \oplus (2G_n)$.*

*Proof.* We start from the sequences $P_{i,j}$ and $G_{i,j}$ defined in Section 4.1.2 corresponding to the Kogge-Stone carry look-ahead adder, and we proceed as in Section 4.1.1. We define the variables:

$$P_i := \sum_{j=2^i-1}^{k-1} 2^j P_{i,j} \quad G_i := \sum_{j=0}^{k-1} 2^j G_{i,j}$$

which from (4.8) gives the initial condition $P_0 = x \oplus y$ and $G_0 = x \wedge y$, and using (4.9):

$$\begin{aligned} P_i &= \sum_{j=2^i-1}^{k-1} 2^j P_{i,j} = \sum_{j=2^i-1}^{k-1} 2^j (P_{i-1,j} \wedge P_{i-1,j-2^{i-1}}) \\ &= \left( \sum_{j=2^i-1}^{k-1} 2^j P_{i-1,j} \right) \wedge \left( \sum_{j=2^i-1}^{k-1} 2^j P_{i-1,j-2^{i-1}} \right) \end{aligned}$$

We can start the summation of the $P_{i,j}$ bits with $j = 2^{i-1} - 1$ instead of $2^i - 1$, because the other summation still starts with $j = 2^i - 1$, hence the corresponding bits are ANDed with 0. This gives:

$$\begin{aligned} P_i &= \left( \sum_{j=2^{i-1}-1}^{k-1} 2^j P_{i-1,j} \right) \wedge \left( \sum_{j=2^i-1}^{k-1} 2^j P_{i-1,j-2^{i-1}} \right) \\ &= P_{i-1} \wedge \left( \sum_{j=2^{i-1}-1}^{k-1-2^{i-1}} 2^{j+2^{i-1}} P_{i-1,j} \right) = P_{i-1} \wedge (P_{i-1} \ll 2^{i-1}) \end{aligned}$$

Hence we get the same recursion formula for $P_i$ as in (4.10). Similarly we have using (4.9):

$$\begin{aligned} G_i &= \sum_{j=0}^{k-1} 2^j G_{i,j} = \sum_{j=2^{i-1}}^{k-1} 2^j \left( (P_{i-1,j} \wedge G_{i-1,j-2^{i-1}}) \oplus G_{i-1,j} \right) + \sum_{j=0}^{2^{i-1}-1} 2^j G_{i-1,j} \\ &= \left( \sum_{j=2^{i-1}}^{k-1} 2^j \left( P_{i-1,j} \wedge G_{i-1,j-2^{i-1}} \right) \right) \oplus G_{i-1} \\ &= \left( P_{i-1} \wedge (G_{i-1} \ll 2^{i-1}) \right) \oplus G_{i-1} \end{aligned}$$

Therefore we obtain the same recurrence for $P_i$ and $G_i$ as (4.10). Since from Lemma 4.1 we have that $c_{j+1} = G_{i,j}$ for all $2^{i-1} \le j < 2^i$, and $G_{i,j} = G_{i-1,j}$ for $0 \le j < 2^{i-1}$, we obtain $c_{j+1} = G_{i,j}$ for all $0 \le j < 2^i$. Taking $i = n = \lceil \log_2(k-1) \rceil$, we obtain $c_{j+1} = G_{n,j}$ for all $0 \le j \le k-2 < k-1 \le 2^n$. This implies:

$$\sum_{j=0}^{k-1} 2^j c_j = \sum_{j=1}^{k-1} 2^j c_j = 2 \sum_{j=0}^{k-2} 2^j c_{j+1} = 2 \sum_{j=0}^{k-2} 2^j G_{n,j} = 2G_n$$

Since from Lemma 4.1 we have $(x+y)^{(j)} = x^{(j)} \oplus y^{(j)} \oplus c_j$ for all $0 \le j < k$, this implies $x + y = x \oplus y \oplus (2G_n)$ as required. □ □

The complexity of the previous recursion is only $\mathcal{O}(\log k)$, as opposed to $\mathcal{O}(k)$ with Goubin's recursion. The sequence can be computed using the algorithm below; note that we do not compute the last element $P_n$ since it is not used in the computation of $G_n$. Note also that the algorithm below could be used as a $\mathcal{O}(\log k)$ implementation of arithmetic addition $z = x + y \bmod 2^k$ for processors having only Boolean operations.

**Algorithm 21** Kogge-Stone Adder

```
Input: $x, y \in \{0,1\}^k$, and $n = \max(\lceil \log_2(k-1) \rceil, 1)$.
Output: $z = x + y \bmod 2^k$
1: $P \leftarrow x \oplus y$
2: $G \leftarrow x \wedge y$
3: for $i := 1$ to $n-1$ do
4:     $G \leftarrow (P \wedge (G \ll 2^{i-1})) \oplus G$
5:     $P \leftarrow P \wedge (P \ll 2^{i-1})$
6: end for
7: $G \leftarrow (P \wedge (G \ll 2^{n-1})) \oplus G$
8: return $x \oplus y \oplus (2G)$
```

## 4.2 Our New Conversion Algorithm

Our new conversion algorithm from arithmetic to Boolean masking is a direct application of the Kogge-Stone adder in Algorithm 21. We are given as input two arithmetic shares $A$, $r$ of $x = A + r \bmod 2^k$, and we must compute $x'$ such that $x = x' \oplus r$, without leaking information about $x$.

Since Algorithm 21 only contains Boolean operations, it is easy to protect against first-order attacks. Assume that we give as input the two arithmetic shares $A$ and $r$ to Algorithm 21; the algorithm first computes $P = A \oplus r$ and $G = A \wedge r$, and after $n$ iterations outputs $x = A + r = A \oplus r \oplus (2G)$. Obviously one cannot compute $P = A \oplus r$ and $G = A \wedge r$ directly since that would reveal information about the sensitive variable $x = A + r$. Instead we protect all intermediate variables with a random mask $s$ using standard techniques, that is we only work with $P' = P \oplus s$ and $G' = G \oplus s$. Eventually we obtain a masked $x' = x \oplus s$ as required, in time $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$.

### 4.2.1 Secure Computation of AND

Since Algorithm 21 contains AND operations, we first show how to secure the AND operation against first-order attacks. The technique is essentially the same as in [ISW03]. With $x = x' \oplus s$ and $y = y' \oplus t$ for two independent random masks $s$ and $t$, we have for any $u$:

$$(x \wedge y) \oplus u = \big((x' \oplus s) \wedge (y' \oplus t)\big) \oplus u = (x' \wedge y') \oplus (x' \wedge t) \oplus (s \wedge y') \oplus (s \wedge t) \oplus u$$

**Algorithm 22** SecAnd

```
Input: $x'$, $y'$, $s$, $t$, $u$ such that $x' = x \oplus s$ and $y' = y \oplus t$.
Output: $z'$ such that $z' = (x \wedge y) \oplus u$.
1: $z' \leftarrow u \oplus (x' \wedge y')$
2: $z' \leftarrow z' \oplus (x' \wedge t)$
3: $z' \leftarrow z' \oplus (s \wedge y')$
4: $z' \leftarrow z' \oplus (s \wedge t)$
5: return $z'$
```

We see that the SecAnd algorithm requires 8 Boolean operations. The following Lemma shows that the SecAnd algorithm is secure against first-order attacks.

**Lemma 4.2.** *When $s$, $t$ and $u$ are uniformly and independently distributed in $\mathbb{F}_{2^k}$, all intermediate variables in the* SecAnd *algorithm have a distribution independent from $x$ and $y$.*

*Proof.* Since $s$ and $t$ are uniformly and independently distributed in $\mathbb{F}_{2^k}$, the variables $x' = x \oplus s$ and $y' = y \oplus t$ are also uniformly and independently distributed in $\mathbb{F}_{2^k}$. Therefore the distribution of $x' \wedge y'$ is independent from $x$ and $y$. The same holds for the variables $x' \wedge t$, $s \wedge y'$ and $s \wedge t$. Moreover since $u$ is uniformly distributed in $\mathbb{F}_{2^k}$, the distribution of $z'$ from Line 1 to Line 4 is uniform in $\mathbb{F}_{2^k}$; hence its distribution is also independent from $x$ and $y$. □ □

### 4.2.2 Secure Computation of XOR

Similarly we show how to secure the XOR computation of Algorithm 21. With $x = x' \oplus s$ and $y = y' \oplus u$ where $s$ and $u$ are two independent masks, we have:

$$(x \oplus y) \oplus s = x' \oplus s \oplus y' \oplus u \oplus s = x' \oplus y' \oplus u$$

**Algorithm 23** SecXor

```
Input: $x'$, $y'$, $u$, such that $x' = x \oplus s$ and $y' = y \oplus u$.
Output: $z'$ such that $z' = (x \oplus y) \oplus s$.
1: $z' \leftarrow x' \oplus y'$
2: $z' \leftarrow z' \oplus u$
3: return $z'$
```

We see that the SecXor algorithm requires 2 Boolean operations. The following Lemma shows that the SecXor algorithm is secure against first-order attacks. It is easy to see that all the intermediate variables in the algorithm are uniformly distributed in $\mathbb{F}_{2^k}$, and hence the proof is straightforward.

**Lemma 4.3.** *When $s$ and $u$ are uniformly and independently distributed in $\mathbb{F}_{2^k}$, all intermediate variables in the* SecXor *algorithm have a distribution independent from $x$ and $y$.*

### 4.2.3 Secure Computation of Shift

Finally we show how to secure the Shift operation in Algorithm 21 against first-order attacks. With $x = x' \oplus s$, we have for any $t$:

$$(x \ll j) \oplus t = ((x' \oplus s) \ll j) \oplus t = (x' \ll j) \oplus (s \ll j) \oplus t$$

This gives the following algorithm.

**Algorithm 24** SecShift

```
Input: $x'$, $s$, $t$ and $j$ such that $x' = x \oplus s$ and $j > 0$.
Output: $y'$ such that $y' = (x \ll j) \oplus t$.
1: $y' \leftarrow t \oplus (x' \ll j)$
2: $y' \leftarrow y' \oplus (s \ll j)$
3: return $y'$
```

We see that the SecShift algorithm requires 4 Boolean operations. The following Lemma shows that the SecShift algorithm is secure against first-order attacks. The proof is straightforward so we omit it.

**Lemma 4.4.** *When $s$ and $t$ are uniformly and independently distributed in $\mathbb{F}_{2^k}$, all intermediate variables in the* SecShift *algorithm have a distribution independent from $x$.*

### 4.2.4 Our New Conversion Algorithm

Finally we can convert Algorithm 21 into a first-order secure algorithm by protecting all intermediate variables with a random mask; see Algorithm 25 below.

Since the SecAnd subroutine requires 8 operations, the SecXor subroutine requires 2 operations, and the SecShift subroutine requires 4 operations, lines 7 to 11 require $2 \cdot 8 + 2 \cdot 4 + 2 + 2 = 28$ operations, hence $28 \cdot (n-1)$ operations for the main loop. The total number of operations is then $7 + 28 \cdot (n-1) + 4 + 8 + 2 + 4 = 28 \cdot n - 3$. In summary, for a register size $k = 2^n$ the number of operations is $28 \cdot \log_2 k - 3$, in addition to the generation of 3 random numbers. Note that the same random numbers $s$, $t$ and $u$ can actually be used for all executions of the conversion algorithm in a given execution. The following Lemma proves the security of our new conversion algorithm against first-order attacks.

**Lemma 4.5.** *When $r$ is uniformly distributed in $\mathbb{F}_{2^k}$, any intermediate variable in Algorithm 25 has a distribution independent from $x = A + r \bmod 2^k$.*

*Proof.* The proof is based on the previous lemma for SecAnd, SecXor and SecShift, and also the fact that all intermediate variables from Line 2 to 5 and in lines 12, 13, 18, and 19 have a distribution independent from $x$. Namely $(A \oplus t) \wedge r$ and $t \wedge r$ have a distribution independent from $x$, and the other intermediate variables have the uniform distribution. □ □

**Algorithm 25** Kogge-Stone Arithmetic to Boolean Conversion

```
Input: $A, r \in \{0,1\}^k$ and $n = \max(\lceil \log_2(k-1) \rceil, 1)$
Output: $x'$ such that $x' \oplus r = A + r \bmod 2^k$.
 1: Let $s \leftarrow \{0,1\}^k$, $t \leftarrow \{0,1\}^k$, $u \leftarrow \{0,1\}^k$.
 2: $P' \leftarrow A \oplus s$
 3: $P' \leftarrow P' \oplus r$                              ▷ $P' = (A \oplus r) \oplus s = P \oplus s$
 4: $G' \leftarrow s \oplus \big((A \oplus t) \wedge r\big)$
 5: $G' \leftarrow G' \oplus (t \wedge r)$                  ▷ $G' = (A \wedge r) \oplus s = G \oplus s$
 6: for $i := 1$ to $n - 1$ do
 7:     $H \leftarrow \mathsf{SecShift}(G', s, t, 2^{i-1})$          ▷ $H = (G \ll 2^{i-1}) \oplus t$
 8:     $U \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$           ▷ $U = \big(P \wedge (G \ll 2^{i-1})\big) \oplus u$
 9:     $G' \leftarrow \mathsf{SecXor}(G', U, s, u)$           ▷ $G' = \big((P \wedge (G \ll 2^{i-1})) \oplus G\big) \oplus s$
10:     $H \leftarrow \mathsf{SecShift}(P', s, t, 2^{i-1})$          ▷ $H = (P \ll 2^{i-1}) \oplus t$
11:     $P' \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$          ▷ $P' = \big(P \wedge (P \ll 2^{i-1})\big) \oplus u$
12:     $P' \leftarrow P' \oplus s$
13:     $P' \leftarrow P' \oplus u$                          ▷ $P' = \big(P \wedge (P \ll 2^{i-1})\big) \oplus s$
14: end for
15: $H \leftarrow \mathsf{SecShift}(G', s, t, 2^{n-1})$              ▷ $H = (G \ll 2^{n-1}) \oplus t$
16: $U \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$               ▷ $U = \big(P \wedge (G \ll 2^{n-1})\big) \oplus u$
17: $G' \leftarrow \mathsf{SecXor}(G', U, s, u)$               ▷ $G' = \big((P \wedge (G \ll 2^{n-1})) \oplus G\big) \oplus s$
18: $x' \leftarrow A \oplus 2G'$                             ▷ $x' = (A + r) \oplus r \oplus 2s$
19: $x' \leftarrow x' \oplus 2s$                             ▷ $x' = (A + r) \oplus r$
20: return $x'$
```

## 4.3 Addition Without Conversion

Beak and Noh proposed a method to mask the ripple carry adder in [BN05]. Similarly, Karroumi *et al.* [KRJ04] used Goubin's recursion formula (2.1) to compute an arithmetic addition $z = x + y \bmod 2^k$ directly with masked shares $x' = x \oplus s$ and $y' = y \oplus r$, that is without first converting $x$ and $y$ from Boolean to arithmetic masking, then performing the addition with arithmetic masks, and then converting back from arithmetic to Boolean masks. They showed that this can lead to better performances in practice for the block cipher XTEA.

In this section we describe an analogous algorithm for performing addition directly on the masked shares, based on the Kogge-Stone adder instead of Goubin's formula, to get $\mathcal{O}(\log k)$ complexity instead of $\mathcal{O}(k)$. More precisely, we receive as input the shares $x'$, $y'$ such that $x' = x \oplus s$ and $y' = y \oplus r$, and the goal is to compute $z'$ such that $z' = (x + y) \oplus r$. For this it suffices to perform the addition $z = x + y \bmod 2^k$ as in Algorithm 21, but with the masked variables $x' = x \oplus s$ and $y' = y \oplus r$ instead of $x$, $y$, while protecting all intermediate variables with a Boolean mask; this is straightforward since Algorithm 21 contains only Boolean operations; see Algorithm 26 below.

**Algorithm 26** Kogge-Stone Masked Addition

```
Input: $x', y', r, s \in \{0,1\}^k$ and $n = \max(\lceil \log_2(k-1) \rceil, 1)$.
Output: $z'$ such that $z' = (x + y) \oplus r$, where $x = x' \oplus s$ and $y = y' \oplus r$
 1: Let $t \leftarrow \{0,1\}^k$, $u \leftarrow \{0,1\}^k$.
 2: $P' \leftarrow \mathsf{SecXor}(x', y', s, r)$                 ▷ $P' = (x \oplus y) \oplus s = P \oplus s$
 3: $G' \leftarrow \mathsf{SecAnd}(x', y', s, r, u)$              ▷ $G' = (x \wedge y) \oplus u = G \oplus u$
 4: $G' \leftarrow G' \oplus s$
 5: $G' \leftarrow G' \oplus u$                                    ▷ $G' = (x \wedge y) \oplus s = G \oplus s$
 6: for $i := 1$ to $n - 1$ do
 7:     $H \leftarrow \mathsf{SecShift}(G', s, t, 2^{i-1})$       ▷ $H = (G \ll 2^{i-1}) \oplus t$
 8:     $U \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$            ▷ $U = (P \wedge (G \ll 2^{i-1})) \oplus u$
 9:     $G' \leftarrow \mathsf{SecXor}(G', U, s, u)$              ▷ $G' = ((P \wedge (G \ll 2^{i-1})) \oplus G) \oplus s$
10:     $H \leftarrow \mathsf{SecShift}(P', s, t, 2^{i-1})$       ▷ $H = (P \ll 2^{i-1}) \oplus t$
11:     $P' \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$           ▷ $P' = (P \wedge (P \ll 2^{i-1})) \oplus u$
12:     $P' \leftarrow P' \oplus s$
13:     $P' \leftarrow P' \oplus u$                                ▷ $P' = (P \wedge (P \ll 2^{i-1})) \oplus s$
14: end for
15: $H \leftarrow \mathsf{SecShift}(G', s, t, 2^{n-1})$           ▷ $H = (G \ll 2^{n-1}) \oplus t$
16: $U \leftarrow \mathsf{SecAnd}(P', H, s, t, u)$                ▷ $U = (P \wedge (G \ll 2^{n-1})) \oplus u$
17: $G' \leftarrow \mathsf{SecXor}(G', U, s, u)$                  ▷ $G' = ((P \wedge (G \ll 2^{n-1})) \oplus G) \oplus s$
18: $z' \leftarrow \mathsf{SecXor}(y', x', r, s)$                 ▷ $z' = (x \oplus y) \oplus r$
19: $z' \leftarrow z' \oplus (2G')$                               ▷ $z' = (x + y) \oplus 2s \oplus r$
20: $z' \leftarrow z' \oplus 2s$                                  ▷ $z' = (x + y) \oplus r$
21: return $z'$
```

As previously the main loop requires $28 \cdot (n-1)$ operations. The total number of operations is then $12 + 28 \cdot (n-1) + 20 = 28 \cdot n + 4$. In summary, for a register size $k = 2^n$ the number of operations is $28 \cdot \log_2 k + 4$, with additionally the generation of 2 random numbers; as previously those 2 random numbers along with $r$ and $s$ can be reused for subsequent additions within the same execution. The following Lemma proves the security of Algorithm 26 against first-order attacks. The proof is similar to the proof of Lemma 4.5 and is therefore omitted.

**Lemma 4.6.** *For a uniformly and independently distributed randoms $r \in \{0,1\}^k$ and $s \in \{0,1\}^k$, any intermediate variable in the* Kogge-Stone Masked Addition *has the uniform distribution.*

## 4.4 Extension to Higher-order Masking

The first conversion algorithms between Boolean and arithmetic masking secure against $t$-th order attack (instead of first-order only) were presented in [CGV14] (Chapter 3). We first described an algorithm for secure addition modulo $2^k$ directly with $n$ Boolean shares (where $n \geq 2t+1$), with complexity $\mathcal{O}(n^2 \cdot k)$. The algorithm was then used as a subroutine to obtain conversion algorithms in both directions, again with complexity $\mathcal{O}(n^2 \cdot k)$. The algorithms were proven secure in the ISW framework for private circuits [ISW03].

Our improved solution can naturally be extended to secure the addition against attacks of order $t$, where we use $d = 2t+1$ shares (instead of two used for the first-order). The algorithms given in Chapter 3 uses Goubin's formula and hence require time in $\mathcal{O}(d^2 k)$. We can use the similar techniques as in first-order masking and improve the solution to $\mathcal{O}(d^2 \log k)$. Namely, we modify Algorithm 21, where all the operations are performed on $d$ shares. The corresponding algorithm is given in Algorithm 27. Here HOSecAnd is a function which securely computes the Boolean AND on given shares. More precisely, it is the higher-order version of the function SecAnd (same as Algorithm 13). Note that the security of Algorithm 27 directly follows from the ISW scheme [ISW03] similar to the algorithms in [CGV14].

**Algorithm 27** HO-secure Kogge-Stone Masked Addition

```
Input: $(x_i)$ and $(y_i)$ for $1 \leq i \leq d$
Output: $(z_i)$ for $1 \leq i \leq d$, with $\bigoplus_{i=1}^{d} z_i = \bigoplus_{i=1}^{d} x_i + \bigoplus_{i=1}^{d} y_i$
 1: $(P_i)_{1\leq i\leq d} \leftarrow (x_i)_{1\leq i\leq d} \oplus (y_i)_{1\leq i\leq d}$        ▷ $P = x \oplus y$
 2: $(G_i)_{1\leq i\leq d} \leftarrow \mathsf{HOSecAnd}((x_i)_{1\leq i\leq n}, (y_i)_{1\leq i\leq d})$        ▷ $G = x \wedge y$
 3: for $i := 1$ to $n-1$ do
 4:     $(G'_i)_{1\leq i\leq d} \leftarrow \mathsf{HOSecAnd}((P_i)_{1\leq i\leq n}, (G_i)_{1\leq i\leq d} \ll 2^{i-1})$
 5:     $(G_i)_{1\leq i\leq d} \leftarrow (G'_i)_{1\leq i\leq d} \oplus (G_i)_{1\leq i\leq d}$
 6:     $(P_i)_{1\leq i\leq d} \leftarrow \mathsf{HOSecAnd}((P_i)_{1\leq i\leq n}, (P_i)_{1\leq i\leq d} \ll 2^{i-1})$
 7: end for
 8: $(G'_i)_{1\leq i\leq d} \leftarrow \mathsf{HOSecAnd}((P_i)_{1\leq i\leq n}, (G_i)_{1\leq i\leq d} \ll 2^{n-1})$
 9: $(G_i)_{1\leq i\leq d} \leftarrow (G'_i)_{1\leq i\leq d} \oplus (G_i)_{1\leq i\leq d}$
10: $(z_i)_{1\leq i\leq d} \leftarrow (x_i)_{1\leq i\leq d} \oplus (y_i)_{1\leq i\leq d} \oplus 2(G_i)_{1\leq i\leq d}$
11: return $(z_i)_{1\leq i\leq d}$
```

## 4.5 Analysis and Implementation

### 4.5.1 Comparison With Existing Algorithms

We compare in Table 4.1 the complexity of our new algorithms with Goubin's algorithms and Debraize's algorithms for various addition bit sizes $k$.[1] We give the number of random numbers required for each of the algorithms as well as the count

[1]For Debraize's algorithm the operation count does not involve the precomputation module. In case of $k = 8$ and $l = 8$ the result can be obtained just by accessing the lookup table.

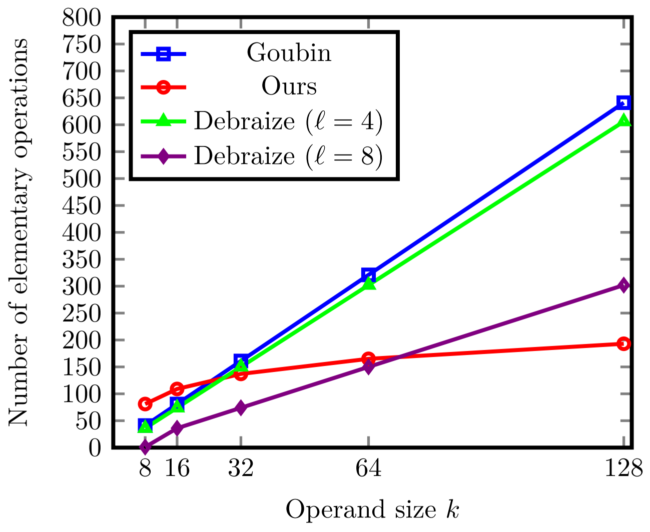


Figure 4.1: Comparison of elementary operations required for Goubin's algorithms, Debraize's algorithm and our new algorithms for various values of $k$.

of required elementary operations. Goubin's original conversion algorithm from arithmetic to Boolean masking required $5k+5$ operations and a single random generation. This was recently improved by Karroumi *et al.* down to $5k+1$ operations [KRJ04]. The authors also provided an algorithm to compute first-order secure addition on Boolean shares using Goubin's recursion formula, requiring $5k+8$ operations and a single random generation (Refer to Section 2.5.2). On the other hand we require $19(k/\ell)-2$ operations for Debraize's algorithm with the lookup table of size $2^l$ and generation of two randoms.

| Algorithm | rand | $k=8$ | $k=16$ | $k=32$ | $k=64$ | $k$ |
|---|---|---|---|---|---|---|
| Goubin's A→B conversion | 1 | 41 | 81 | 161 | 321 | $5k+1$ |
| New A→B conversion | 3 | 81 | 109 | 137 | 165 | $28\log_2 k - 3$ |
| Goubin's addition [KRJ04] | 1 | 48 | 88 | 168 | 328 | $5k+8$ |
| New addition | 2 | 88 | 116 | 144 | 172 | $28\log_2 k + 4$ |
| Debraize's A→B conversion ($\ell = 4$) | 2 | 36 | 74 | 150 | 302 | $19(k/4) - 2$ |
| Debraize's A→B conversion ($\ell = 8$) | 2 | - | 36 | 74 | 150 | $19(k/8) - 2$ |

Table 4.1: Number of randoms (rand) and elementary operations required for Goubin's algorithms, Debraize's algorithm and our new algorithms for various values of $k$.

From Figure 4.1 and Figure 4.2 we see that our algorithms outperform Goubin's algorithms for $k \geq 32$ and performs better or comparable to Debraize's algorithm depending on $\ell$. Moreover, in most of the RISC based microcontrollers (ex: AVR, ARM) shifts are for free, and hence SecShift actually costs only 2 instructions instead of 4 instructions. For example, Line 1 in SecShift will be compiled to:

$$eor\ Rz, Rz, Rs \ll j$$

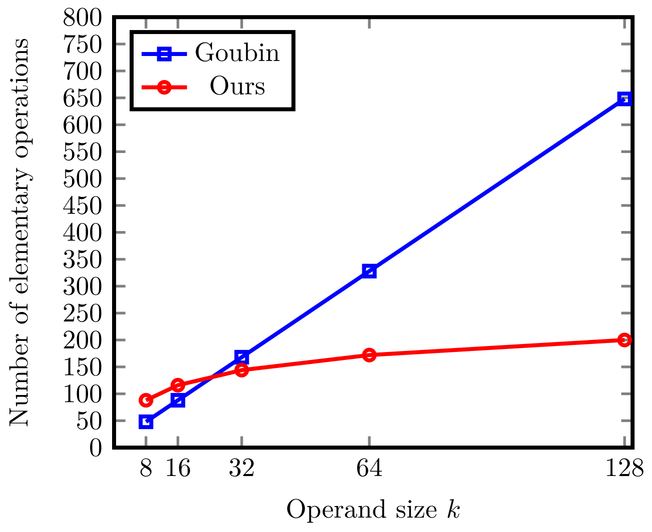


Figure 4.2: Comparison of elementary operations required for Goubin's addition and our addition for various values of $k$.

where $Rz$, $Rs$ are the registers containing $t$ and $x'$, and $j$ is the shift value (a compile-time constant if we use loop-unrolling). This instruction executes in a single clock cycle with or without the shift. If we consider this, our algorithms perform even better in practice as we show in the next section. In practice, most cryptographic constructions performing arithmetic operations use addition modulo $2^{32}$; for example HMAC-SHA-1 [NIS95] and XTEA [NW97]. There also exists cryptographic constructions with additions modulo $2^{64}$, for example Threefish used in the hash function Skein, a SHA-3 finalist, and the SPECK block-cipher (see Section 4.7).

### 4.5.2 Practical Implementation

We have implemented our new algorithms along with Goubin's algorithms; we have also implemented the table-based arithmetic to Boolean conversion algorithm described by Debraize in [Deb12]. For Debraize's algorithm, we considered two possibilities for the partition of the data, with word length $\ell = 4$ and $\ell = 8$. Our implementations were done on a 32-bit AVR microcontroller *AT32UC3A0512*, based on RISC microprocessor architecture. It can run at frequencies up to 66 MHZ and has SRAM of size 64 KB along with a flash of 512 KB. We used the C programming language and the machine code was produced using the AVR-GCC compiler with further optimization (*e.g.* loop unrolling). For generation of random numbers we used a pseudorandom number generator based on linear feedback shift registers. [2]

The results are summarized in Table 4.2. From Figure 4.3 and Figure 4.4 we see that our new algorithms perform better than Goubin's algorithms from $k = 32$

[2]Note that the reported results have strong dependency on the used RNG and hence can change if a different RNG is used.

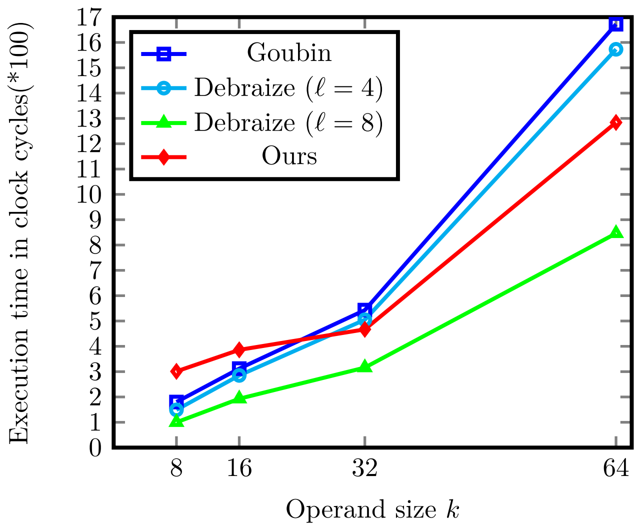


Figure 4.3: Number of clock cycles required for Goubin's conversion algorithm, Debraize's conversion algorithm, our new conversion algorithm for various values of $k$.

onward. When $k = 32$, our algorithms perform roughly 14% better than Goubin's algorithms. Moreover, our conversion algorithm performs 7% better than Debraize's algorithm ($\ell = 4$). For $k = 64$, we can see even better improvement i.e., 23% faster than Goubin's algorithm and 22% better than Debraize's algorithm ($\ell = 4$). On the other hand, Debraize's algorithm performs better than our algorithms for $\ell = 8$ ; however as opposed to Debraize's algorithm our conversion algorithm requires neither preprocessing nor extra memory.

| | $k = 8$ | $k = 16$ | $k = 32$ | $k = 64$ | Prep. | Mem. |
|---|---|---|---|---|---|---|
| Goubin's A→B conversion | 180 | 312 | 543 | 1672 | - | - |
| Debraize's A→B conversion ($\ell = 4$) | 149 | 285 | 505 | 1573 | 1221 | 32 |
| Debraize's A→B conversion ($\ell = 8$) | - | 193 | 316 | 846 | 18024 | 1024 |
| New A→B conversion | 301 | 386 | 467 | 1284 | - | - |
| Goubin's addition [KRJ04] | 235 | 350 | 582 | 1789 | - | - |
| New addition | 344 | 429 | 513 | 1340 | - | - |

Table 4.2: Number of clock cycles on a 32-bit processor required for Goubin's conversion algorithm, Debraize's conversion algorithm, our new conversion algorithm, Goubin's addition from [KRJ04], and our new addition, for various arithmetic sizes $k$. The last two columns denote the precomputation time and the table size (in bytes) required for Debraize's algorithm.

We give the results of our higher-order secure addition (Algorithm 27) in Table 4.3. We compare our implementation with that of Coron-Großschädl-Vadnala [CGV14] for security order $t = 2, 3$. From Figure 4.5 we see that our algorithms significantly improve the execution time (up to 70%).

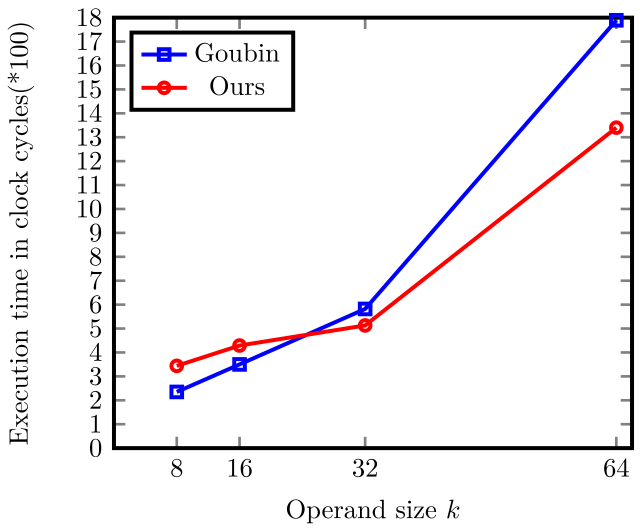


Figure 4.4: Number of clock cycles required for Goubin's addition from [KRJ04], and our new addition, for various values of $k$.

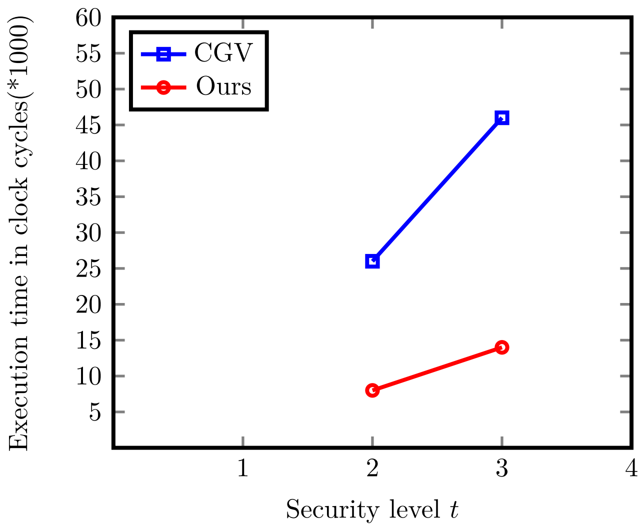


Figure 4.5: Comparison of running time requirements for CGV [CGV14] algorithm and our new algorithm for $t = 2, 3$.

| Algorithm | $t = 2$ | | $t = 3$ | |
|---|---|---|---|---|
| | Time | Rand | Time | Rand |
| CGV [CGV14] | 26 | 320 | 46 | 672 |
| Our algorithms | 8 | 100 | 14 | 210 |

Table 4.3: Time (in thousands of cycles) and number of randoms (Rand) required for CGV [CGV14] algorithm and our new algorithm for $t = 2, 3$.

## 4.6 Application to HMAC-SHA-1

In this section, we apply our countermeasure to obtain a first-order secure implementation of HMAC-SHA-1. SHA-1 is a cryptograhic hash function designed by NSA that is still widely used in numerous commercial applications. As SHA-1 involves both modular addition and XOR operations, we must either convert between Boolean and arithmetic masking, or perform the arithmetic additions directly on the Boolean shares as suggested in [KRJ04].

### 4.6.1 HMAC-SHA-1

SHA-1 processes the input in blocks of 512-bits and produces a message digest of 160 bits. If the length of the message is not a multiple of 512, the message is appended with zeros, followed by a "1". The last 64 bits of the message contains the length of the original message in bits. All operations are performed on 32-bit words, producing an output of five words. The initial values of the five hash words are: `H0 = 0x67452301, H1 = 0xEFCDAB89, H2 = 0x98BADCFE, H3 = 0x10325476, H4 = 0xC3D2E1F0`. Each 512-bit message block $M[0], \cdots, M[15]$ is expanded to 80 words as follows:

$$W[i] = \begin{cases} M[i] & \text{if } i \leq 15 \\ (W[i-3] \oplus W[i-8] \oplus W[i-14] \oplus W[i-16]) \lll 1 & \text{Otherwise} \end{cases}$$

One initially lets $A = H_0$, $B = H_1$, $C = H_2, D = H_3$ and $E = H_4$, where the $H_i$'s are initial constants. The main loop is defined as follows, for $i = 0$ to $i = 79$.

$$\begin{aligned} Temp &= (A \lll 5) + f(i, B, C, D) + E + W[i] + k[i] \\ E &= D;\ D = C;\ C = B \lll 30;\ B = A;\ A = Temp \end{aligned}$$

where the function $f$ is defined as:

$$f(i, B, C, D) = \begin{cases} (B \wedge C) \vee ((\neg B) \wedge D) & \text{if } 0 \leq i \leq 19 \\ (B \oplus C \oplus D) & \text{if } 20 \leq i \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{if } 40 \leq i \leq 59 \\ (B \oplus C \oplus D) & \text{if } 60 \leq i \leq 79 \end{cases}$$

and the constant $k$ is defined as:

$$k[i] = \begin{cases} \texttt{0x5A827999} & \text{if } 0 \leq i \leq 19 \\ \texttt{0x6ED9EBA1} & \text{if } 20 \leq i \leq 39 \\ \texttt{0x8F1BBCDC} & \text{if } 40 \leq i \leq 59 \\ \texttt{0xCA62C1D6} & \text{if } 60 \leq i \leq 79 \end{cases}$$

After the main loop one lets:

$$H_0 \leftarrow H_0+A, \;\; H_1 \leftarrow H_1+B, \;\; H_2 \leftarrow H_2+C, \;\; H_3 \leftarrow H_3+C, \;\; H_4 \leftarrow H_4+D \quad (4.11)$$

and one processes the next block. All additions are performed modulo $2^{32}$. Eventually the final hash result is the 160-bit string $H_0\|H_1\|H_2\|H_3\|H_4$.

HMAC-SHA-1 of a message $M$ is computed as:

$$H\big(K \oplus opad \;\|\; H(K \oplus ipad, M)\big)$$

where $H$ is the SHA-1 function, $K$ is the secret key, and *ipad* and *opad* are constants.

### 4.6.2 First-order Secure HMAC-SHA-1

In this section we show how to protect HMAC-SHA-1 against first-order attacks, using either Goubin's Boolean to arithmetic conversion (Section 2.5.1) and our new arithmetic to Boolean conversion (Algorithm 25), or the addition with Boolean masks (Algorithm 26).

In this section, we apply the algorithms proposed in Section 4.3 and Section 4.2 to obtain a first-order secure HMAC-SHA-1. We present a method to securely compute $H(K \oplus ipad, M)$, which can be used for the final output as well i.e., $H((K \oplus opad)||H(K \oplus ipad, M))$.

**Randomizing the inputs**

The key $K$ is masked with a 64 byte random number $r_1$. The two shares corresponding to the key are: $K_0 = r_1 \oplus K, K_1 = r_2$. Similarly the message $M$ is masked with a random number $r_2$ with the corresponding shares being $M_0 = M \oplus r_2, M_1 = r_2$. After applying XOR with *ipad*, the input to the hash function becomes: $(r_1 \oplus K \oplus ipad, r_1)||(r_2, M \oplus r_2)$. Initial shares of the digest are:

$$\texttt{H0}_0 = \texttt{0x67452301},\; \texttt{H0}_1 = 0,$$

$$\texttt{H1}_0 = \texttt{0xEFCDAB89},\; \texttt{H1}_1 = 0,$$

$$\texttt{H2}_0 = \texttt{0x98BADCFE},\; \texttt{H2}_1 = 0,$$

$$\texttt{H3}_0 = \texttt{0x10325476},\; \texttt{H3}_1 = 0,$$

$$\texttt{H4}_0 = \texttt{0xC3D2E1F0},\; \texttt{H4}_1 = 0.$$

These values will be updated after the processing of every block. Finally, the shares of the 80 words $W[0], \cdots, W[80]$: $W_j[0], \cdots, W_j[80]$ for $j = 0, 1$ are computed as follows:

$$W_j[i] = \begin{cases} M_j[i] & \text{if } i \leq 15 \\ (W_j[i-3] \oplus W_j[i-8] \oplus W_j[i-14] \oplus W_j[i-16]) \lll 1 & \text{Otherwise} \end{cases}$$

### Securing function $f$

The shares of $A, B, C, D, E$ are set to the shares of $H_0, H_1, H_2, H_3, H_4$ correspondingly. The function $f$ computes three different values depending on the index $i$. For $20 \leq i \leq 39$ and $60 \leq i \leq 79$, $f(i, B, C, D) = (B \oplus C \oplus D)$. This can be easily secured by applying XOR on individual shares, i.e.:

$$f(i, B_0, C_0, D_0, , B_1, C_1, D_1) = (B_0 \oplus C_0 \oplus D_0, B_1 \oplus C_1 \oplus D_1)$$

For $0 \leq i \leq 19$, $f(i, B, C, D) = (B \wedge C) \vee ((\neg B) \wedge D)$. We know that

$$\neg(a \oplus b) = (\neg a) \oplus b = a \oplus (\neg b)$$

Hence $(\neg B)$ can be easily obtained as: $(\neg B_0) \oplus B_1$. We can compute $(B \wedge C)$ and $((\neg B) \wedge D)$ securely using the SecAnd function. To compute $a \vee b$ we use the following relation:

$$a \vee b = \neg((\neg a) \wedge (\neg b)) \tag{4.12}$$

For $40 \leq i \leq 59$, $f(i, B, C, D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$. This can be computed securely using the SecAnd function and Equation 4.12.

### Securing the main loop

The main loop consists of the following operation:

$$Temp = (A \ll 5) + f(i, B, C, D) + E + W[t] + k[t]$$

If we use the masked addition method proposed in Section 4.3, we need to perform four additions on the Boolean shares to evaluate the above expression. Since the SHA-1 loop consists of 80 iterations, we need $4 \cdot 80 = 320$ calls to the masked addition (Algorithm 26) for the main loop. Additionally, the update of the $H_i$'s (4.11) needs five calls. Hence, we need a total of 325 calls per each message block . Alternatively, we can use conversion methods to obtain the same result as follows:

1. Convert the Boolean shares to corresponding arithmetic shares using Goubin's algorithm.

2. Perform addition directly on the Boolean shares.

3. Convert the resulting arithmetic shares to Boolean shares using Algorithm 25.

With this approach we need 5 Boolean to arithmetic conversions and 1 arithmetic to Boolean conversion for each iteration. Hence with the additional update of the $H_i$'s we need a total of $5 \cdot 80 + 5 = 405$ Boolean to arithmetic conversions and $80 + 5 = 85$ arithmetic to Boolean conversions.

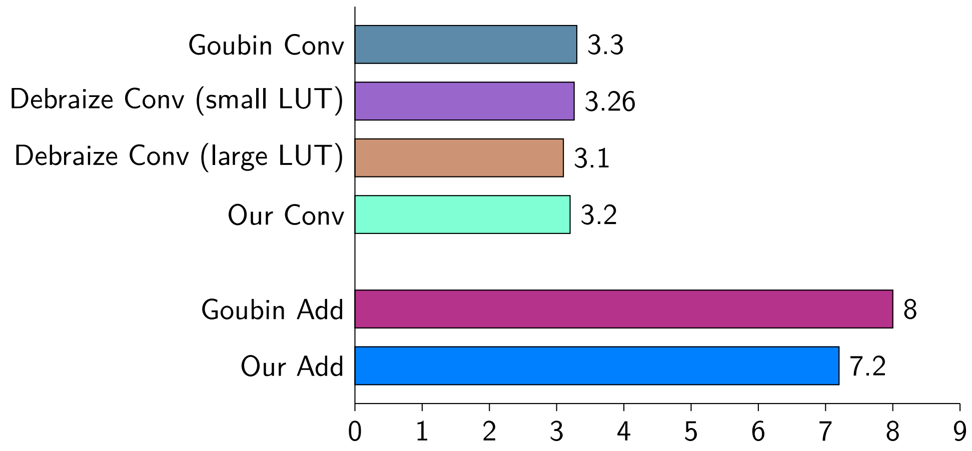


Figure 4.6: Comparison of penalty factor for HMAC-SHA-1 on a 32-bit processor

### 4.6.3 Practical Implementation

We have implemented HMAC-SHA-1 using the technique above on the same microcontroller as in Section 4.5.2. To convert from arithmetic to Boolean masking, we used one of the following: Goubin's algorithm, Debraize's algorithm or our new algorithm. The results for computing HMAC-SHA-1 of a single message block are summarized in Table 4.4. For Debraize's algorithm, the timings also include the precomputaion time required for creating the tables. From Figure 4.6 we see that our algorithms give better performances than Goubin and Debraize ($\ell = 4$), but Debraize with $\ell = 8$ is still slightly better; however as opposed to Debraize, our algorithms do not require extra memory. For the masked addition (instead of conversions), the new algorithm performs 7% better than Goubin's algorithm.

| | Time | Penalty Factor | Memory |
|---|---|---|---|
| HMAC-SHA-1 unmasked | 128 | 1 | - |
| HMAC-SHA-1 with Goubin's conversion | 423 | 3.3 | - |
| HMAC-SHA-1 with Debraize's conversion ($\ell = 4$) | 418 | 3.26 | 32 |
| HMAC-SHA-1 with Debraize's conversion ($\ell = 8$) | 402 | 3.1 | 1024 |
| HMAC-SHA-1 with new conversion | 410 | 3.2 | - |
| HMAC-SHA-1 with Goubin's addition [KRJ04] | 1022 | 8 | - |
| HMAC-SHA-1 with new addition | 933 | 7.2 | - |

Table 4.4: Running time in thousands of clock-cycles and penalty factor for HMAC-SHA-1 on a 32-bit processor. The last column denotes the table size (in bytes) required for Debraize's algorithm.

## 4.7 Application to SPECK

SPECK is a family of lightweight block ciphers proposed by NSA, which provides high throughput for application in software [BSS+13]. The SPECK family includes various ciphers based on ARX (Addition, Rotation, XOR) design with different block and key sizes. To verify the performance results of our algorithms for $k =$

64, we used SPECK 128/128, where block and key sizes both equal to 128 and additions are performed modulo $2^{64}$. The 128-bit key is expanded to 32 64-bit round keys (equal to the number of rounds) using Key Expansion procedure. Each round operates on 128-bit data and consists of following operations: Addition modulo $2^{64}$, Rotate 8 bits right, Rotate 3 bits left and XOR. More precisely, given $x[2i]$, $x[2i+1]$ as input (with each of them 64-bit long), the round $i$ does the following:

$$x[2i+2] = (\mathsf{RotateRight}(x[2i], 8) + x[2i+1]) \oplus key[i]$$
$$x[2i+3] = (\mathsf{RotateLeft}(x[2i+1], 3)) \oplus x[2i+2]$$

Similar to HMAC-SHA-1, we applied all the algorithms to protect SPECK 128/128 against first-order attacks. If we use conversion algorithms, we need to perform two Boolean to arithmetic conversions and one arithmetic to Boolean conversion per round; hence 64 Boolean to arithmetic conversions and 32 arithmetic to Boolean conversions overall. On the other hand, when we perform addition directly on the Boolean shares, we need 32 additions in total. We summarize the performance of all the algorithms in Table 4.5.

| | Time | Penalty Factor | Memory |
|---|---|---|---|
| SPECK unmasked | 2047 | 1 | - |
| SPECK with Goubin's conversion | 63550 | 31 | - |
| SPECK with Debraize's conversion ($\ell = 4$) | 61603 | 30 | 32 |
| SPECK with Debraize's conversion ($\ell = 8$) | 37718 | 18 | 1024 |
| SPECK with new conversion | 51134 | 24 | - |
| SPECK with Goubin's addition [KRJ04] | 62942 | 30 | - |
| SPECK with new addition | 48574 | 23 | - |

Table 4.5: Running time in clock-cycles and penalty factor for SPECK on a 32-bit processor. The last column denotes the table size (in bytes) required for Debraize's algorithm.

As we can see from Figure 4.7 our algorithms outperform Goubin and Debraize's algorithm ($\ell = 4$) similar to HMAC-SHA-1. The results are also better than Debraize's algorithm for ($\ell = 8$), which was not the case for HMAC-SHA-1. Hence, we can conclude that our proposed algorithms provide better performance than all other algorithms for $k = 64$, despite not using any extra memory for the tables.

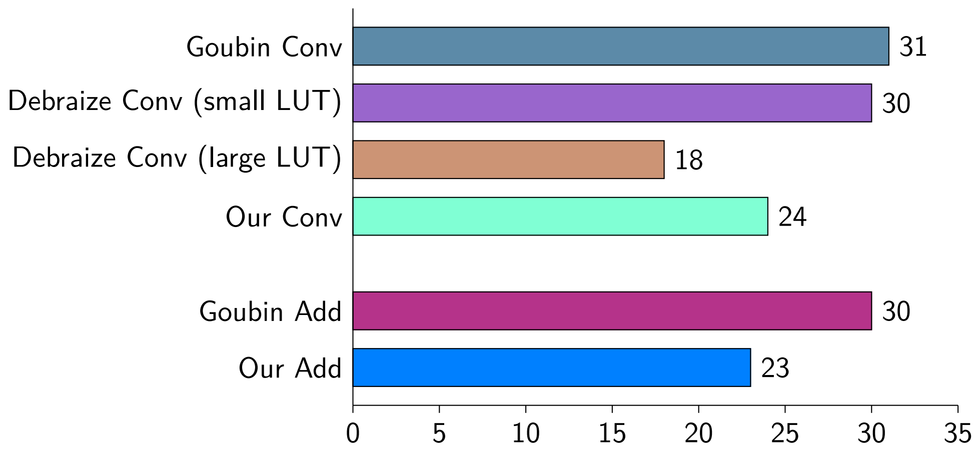


Figure 4.7: Comparison of penalty factor for SPECK-64 on a 32-bit processor

# Chapter 5

# Faster Mask Conversion with Lookup Tables

The algorithms given in Chapter 3 and Chapter 4 require $d = 2t+1$ shares to protect against attacks of order $t$. In this chapter, we improve the algorithms for second-order conversion using lookup tables so that only three shares instead of five are needed, which is the minimal number for second-order security. We first introduce two algorithms to convert from Boolean to arithmetic masking based on the second-order provably secure S-box output computation method proposed by Rivain et al at FSE 2008 (recalled in Chapter 2). The same can be used to obtain second-order secure arithmetic to Boolean masking. Though these algorithms are secure (as we show), they become infeasible for implementation on low-resource devices like smart cards for $n > 10$ (the additions are performed modulo $2^n$), as we require lookup table of size $2^n$. We then show how we can overcome this challenge using divide and conquer approach. Furthermore, we also propose a first-order secure addition algorithm again using lookup tables. This new algorithm gives similar performance compared to the solution of Karroumi-Richard-Joye. We prove the security of all proposed algorithms on the basis of well established assumptions and models. Finally, we provide experimental evidence of our improved mask conversion applied to HMAC-SHA-1. Our results show that the proposed second-order algorithms improve the execution time by 85% compared to the methods given in Chapter 3 and do so with negligible memory overhead. This is a joint work with Johann Großschädl. A part of this work appeared in the proceedings of SPACE, 2013 [CGVT15] and COSADE, 2015 [VG15].

**Contents**

## 5.1 Second-order Boolean to Arithmetic Masking

This section addresses the problem of securely converting second-order Boolean shares to the corresponding arithmetic shares without any second-order or first-order leakage. To start with, we are given three Boolean shares $x_1$, $x_2$, $x_3$ such that $x = x_1 \oplus x_2 \oplus x_3$ where $x$ is a sensitive variable. The goal is to find three arithmetic shares $A_1$, $A_2$, $A_3$ satisfying $x = A_1 + A_2 + A_3$ without leaking any information exploitable in a first or second-order DPA attack. We propose two algorithms to achieve this goal; one is based on Algorithm 1 and the second on Algorithm 2. Both of our algorithms use the secure S-box output computation of Rivain et al [RDP08], which simplifies the security proofs.

The first of our variants is given in Algorithm 28; we devised this conversion by modifying Algorithm 1 appropriately. The algorithm generates two shares $A_2$, $A_3$ randomly from $[\,0, 2^n - 1]$ and computes the third share via the relation $A_1 = (x - A_2) - A_3$. The aim of Algorithm 1 was to output $S(x) \oplus y_1 \oplus y_2$ as result. Hence, it computed $(S(x_1 \oplus a) \oplus y_1) \oplus y_2$ for every possible value of the variable $a$ from 0 to $2^n - 1$, and then obtained the correct value for the case $a = x_2 \oplus x_3$. But here, our aim is to compute $(x - A_2) - A_3$, which requires us to modify the table entries to $((x_1 \oplus a) - A_2) - A_3$ so that we can obtain the correct value when $a = x_2 \oplus x_3$. Note that the subtractions are modulo $2^n$.

**Correctness:** When $a' = r$, $a$ becomes $r \oplus r' = x_2 \oplus x_3$. Thus, $T[a'] = T[r] = ((((x_1 \oplus x_2) \oplus x_3) - A_2) - A_3) = (x - A_2) - A_3$, from which follows that $A_1 = (x - A_2) - A_3$ and finally $x = A_1 + A_2 + A_3$.

**Algorithm 28** Sec2OB→A: First Variant

```
Input: Boolean shares: $x_1 = x \oplus x_2 \oplus x_3$, $x_2$, $x_3$
Output: Arithmetic shares: $A_1 = (x - A_2) - A_3$, $A_2$, $A_3$
1: Randomly generate $n$-bit numbers $r$, $A_2$, $A_3$
2: $r' \leftarrow (r \oplus x_2) \oplus x_3$
3: for $a = 0$ to $2^n - 1$ do
4:     $a' \leftarrow a \oplus r'$
5:     $T[a'] \leftarrow ((x_1 \oplus a) - A_2) - A_3$
6: end for
7: $A_1 = T[r]$
8: return $A_1, A_2, A_3$
```

We devised Algorithm 29 by appropriately adapting Algorithm 2. Again, we first compute the value of $((x_1 \oplus a) - A_2) - A_3$ for all possible values of $a$ and store the result in $R_b$ or $R_{\bar{b}}$, depending on the return value of $compare_b$. When $a = x_2 \oplus x_3$, the value of $x_2 \oplus a$ and $x_3$ become equal, hence $compare_b$ returns $b$. Consequently, the correct value of $A_1 = (x - A_2) - A_3$ is stored in $R_b$. In all other cases (i.e. $a \neq x_2 \oplus x_3$), the value $((x_1 \oplus a) - A_2) - A_3$ is stored in $R_{\bar{b}}$.

**Algorithm 29** Sec2OB→A: Second Variant

```
Input: Boolean shares: $x_1 = x \oplus x_2 \oplus x_3$, $x_2$, $x_3$
Output: Arithmetic shares: $A_1 = (x - A_2) - A_3$, $A_2$, $A_3$
1: Randomly generate $n$-bit numbers $A_2$, $A_3$
2: Randomly generate one bit $b$
3: for $a = 0$ to $2^n - 1$ do
4:     $cmp \leftarrow compare_b(x_2 \oplus a, x_3)$
5:     $R_{cmp} \leftarrow ((x_1 \oplus a) - A_2) - A_3$
6: end for
7: $A_1 = R_b$
8: return $A_1, A_2, A_3$
```

## 5.2 Second-order Arithmetic to Boolean Masking

In this section, we briefly introduce two algorithms to securely convert second-order arithmetic shares into the "corresponding" Boolean shares, whereby the conversion does not introduce any second-order (or first-order) leakage. More precisely, given three arithmetic shares $A_1$, $A_2$, $A_3$ of a sensitive variable $x$ such that $x = A_1 + A_2 + A_3$, both of these algorithms compute the Boolean shares $x_1$, $x_2$, $x_3$ satisfying $x = x_1 \oplus x_2 \oplus x_3$ without second or first-order leakage.

Algorithm 30 employs a lookup table similar to Algorithm 28. Here, the value of $r'$ is $(A_2 - r) + A_3$, where $r$ is a random value in the range $[0, 2^n - 1]$. The table entries corresponding to $a' = a - r'$ are now $((A1 + a) \oplus x_2) \oplus x_3$ instead of $((x_1 \oplus a) - A_2) - A_3$. Similar to Algorithm 28, the two shares $x_2$ and $x_3$ are generated randomly from $[0, 2^n - 1]$, while the third share $x_1$ is $T[r]$.

**Algorithm 30** Sec2OA→B: First Variant

```
Input: Arithmetic shares: $A_1 = (x - A_2) - A_3$, $A_2$, $A_3$
Output: Boolean shares: $x_1 = x \oplus x_2 \oplus x_3$, $x_2$, $x_3$
1: Randomly generate $n$-bit numbers $r$, $x_2$, $x_3$
2: $r' \leftarrow (A_2 - r) + A_3$
3: for $a = 0$ to $2^n - 1$ do
4:     $a' \leftarrow a - r'$
5:     $T[a'] \leftarrow ((A_1 + a) \oplus x_2) \oplus x_3$
6: end for
7: $x_1 = T[r]$
8: return $x_1, x_2, x_3$
```

**Correctness:** When $a' = r$, $a$ becomes $r + r' = A_2 + A_3$. Thus, $T[a'] = T[r] = ((((A_1+A_2)+A_3)\oplus x_2)\oplus x_3) = (x\oplus x_2)\oplus x_3$, from which follows that $x_1 = (x\oplus x_2)\oplus x_3$ and finally $x = x_1 \oplus x_2 \oplus x_3$.

Algorithm 31 shows the other method to convert arithmetic shares of second order to "equivalent" Boolean shares. Among the three Boolean shares, $x_2$ and $x_3$ are generated randomly within the range $[0, 2^{n-1}]$. One of the two registers $R_0$, $R_1$ serves to store the correct value of $x_1$ and the other is used for storing the incorrect value. The compare instruction compares $(a - A_2)$ with $A3$; when they are equal, $compare_b$ returns $b$ and, thus, the result is stored in $R_b$. In this case, the result is the correct value of $x_1$, which means $((A_1 + A_2 + A_3) \oplus x_2) \oplus x_3 = (x \oplus x_2) \oplus x_3$. Otherwise, the result is incorrect and stored in $R'_b$.

**Algorithm 31** Sec20A→B: Second Variant

**Input:** Arithmetic shares: $A_1 = (x - A_2) - A_3$, $A_2$, $A_3$
**Output:** Boolean shares: $x_1 = x \oplus x_2 \oplus x_3$, $x_2$, $x_3$

```
1: Randomly generate $n$-bit numbers $x_2$, $x_3$
2: Randomly generate one bit $b$
3: for $a = 0$ to $2^n - 1$ do
4:     $cmp \leftarrow compare_b(a - A_2, A_3)$
5:     $R_{cmp} \leftarrow ((A_1 + a) \oplus x_2) \oplus x_3$
6: end for
7: $x_1 = R_b$
8: return $x_1, x_2, x_3$
```

## 5.3 Security Analysis

We first review the security model introduced in [RDP08]. Then, based on the same model, we present the security proofs of all our four algorithms against second-order attacks. We assume that the device leaks in the Hamming weight model in which the leakage is proportional to the Hamming weight of the processed data. Below we summarize some basic definitions and results that are used in the proofs for quick reference (taken from [RDP08]).

- *Sensitive variable:* An intermediate variable obtained by applying a certain function on a known value (e.g. plaintext) and the secret key.
- *Primitive random variable:* An intermediate variable generated by a uniform random number generator.
- *Functional dependence:* If an intermediate variable is obtained by applying a discrete function on some other variable $X$, then it is said to be functionally dependent on $X$. Otherwise, it is functionally independent.
- *Statistical dependence:* If the statistical distribution of an intermediate variable varies according to some other variable $X$, then it is said to be statistically

dependent on $X$. Otherwise, it is said to be statistically independent.

- Functional independence implies statistical independence but not vice-versa.
- In second order SCA, leakages from a maximum of two intermediate variables is allowed to be exploited simultaneously. Hence, for a cryptographic algorithm to be called second order secure, it is imperative that every pair of intermediate variables is statistically independent of any sensitive variable.
- A set of intermediate variables is statistically independent of a variable $X$, if and only if all the intermediate variables belonging to the set are statistically independent of $X$.
- For two sets $A$ and $B$, $A \times B$ is statistically independent of a variable $X$, if and only if all the pairs in $A \times B$ are statistically independent of $X$.

  **Lemma 5.1.** *For two statistically independent random variables $X$ and $Y$, it holds that for every measurable function $f$, $f(X)$ is statistically independent of $Y$.*

  **Lemma 5.2.** *For two statistically independent random variables $X$ and $Y$, where $Y$ is uniformly distributed, and for a variable $Z$ statistically independent of $Y$ and functionally independent of $X$, it holds that the pair $(Z, X \oplus Y)$ is statistically independent of $X$.*

**Limitations of the Security Proofs:**

The algorithms in [RDP08], though proven secure against "standard" DPA attacks, suffer from two problems. Firstly, the algorithm not using table computations, i.e. Algorithm 2, is only secure in the Hamming weight model. At COSADE 2012, Coron et al have shown that the algorithm is *not* secure when the device leaks in the Hamming distance model [CGP$^+$12b]. They also demonstrated that a straightforward conversion of a proof from the Hamming weight model to Hamming distance model by initializing the bus (resp. register) with 0 before every write operation has a second-order flaw. As a consequence, the proof of Algorithm 2 given in [RDP08] not valid anymore in the Hamming distance model. The conversion of a security proof from one leakage model to another is still an open issue. Since Algorithm 29 and Algorithm 31 are similar to Algorithm 2, they suffer from said limitation too. However, a solution to the conversion problem for Rivain et al's generic countermeasure for secure S-box computation would, of course, also apply to our algorithms.

Secondly, recent developments in side-channel cryptanalysis have shown that so-called *horizontal side channel attacks* can still defeat the masking [PHL09, TWO13]. By attacking the masked table generation of the algorithms, the attacker can still recover the secret key when the SNR (Signal to noise ratio) is low. But these attacks are generic in the sense that they can be applied to almost all the masking schemes that are used currently. So, our algorithms are no exception to this. The problem of securely generating the masked table is still an open challenge and needs immediate attention. The solution to this problem can be readily applied to our algorithms as well making them secure against these attacks too. Hence, despite these limitations the algorithms proposed here are still relevant to the cryptology community.

**Theorem 5.1.** *Algorithm 28 is secure against second order DPA.*

*Proof.* We follow the same notation as in [RDP08] for the sake of simplicity. Each intermediate variable of the algorithm can be seen as a result of applying the function $I_j$ on the loop index $a$. Assume that $I_{index} = I_{index}(a)$ for $0 \leq a \leq 2^n - 1$ and $I = \bigcup_{index=0}^{num} I_{index}$, where $num$ is the total number of intermediate variables. We list all the intermediate variables used in Algorithm 28 in Table 5.1.

Table 5.1: Intermediate variables used in Algorithm 28

| index | $I_{index}$ |
|---|---|
| 1 | $x_2$ |
| 2 | $x_3$ |
| 3 | $A_2$ |
| 4 | $A_3$ |
| 5 | $r$ |
| 6 | $r \oplus x_2$ |
| 7 | $r \oplus x_2 \oplus x_3$ |
| 8 | $a$ |
| 9 | $a \oplus r \oplus x_2 \oplus x_3$ |
| 10 | $x \oplus x_2 \oplus x_3$ |
| 11 | $x \oplus x_2 \oplus x_3 \oplus a$ |
| 12 | $(x \oplus x_2 \oplus x_3 \oplus a) - A_2$ |
| 13 | $((x \oplus x_2 \oplus x_3 \oplus a) - A_2) - A_3$ |
| 14 | $(x - A_2) - A_3$ |

Table 5.2: Intermediate variables used in Algorithm 29

| index | $I_{index}$ |
|---|---|
| 1 | $x_2$ |
| 2 | $x_3$ |
| 3 | $A_2$ |
| 4 | $A_3$ |
| 5 | $b$ |
| 6 | $a$ |
| 7 | $x_2 \oplus a$ |
| 8 | $\delta_0(x_2 \oplus a \oplus x_3) \oplus b$ |
| 9 | $x \oplus x_2 \oplus x_3$ |
| 10 | $x \oplus x_2 \oplus x_3 \oplus a$ |
| 11 | $(x \oplus x_2 \oplus x_3 \oplus a) - A_2$ |
| 12 | $((x \oplus x_2 \oplus x_3 \oplus a) - A_2) - A_3$ |
| 13 | $(x - A_2) - A_3$ |

We recall that for an algorithm to be secure against second order DPA, no pair of intermediate variables should be statistically dependent on a sensitive variable. So, we need to prove that $I \times I$ is statistically independent of $x$. For simplicity, we divide the set of intermediate variables into three subsets: $E_1 = I_1 \cup I_2 \cup ... \cup I_9$, $E_2 = I_{10} \cup I_{11} \cup ... \cup I_{13}$, $E_3 = I_{14}$. The objective now is to prove that all the combinations of these sets are statistically independent of $x$.

1. $E_1 \times E_1$**:** All the intermediate variables in $E_1$ are functionally independent of $x$. Hence, $E_1 \times E_1$ is statistically independent of $x$.

2. $E_2 \times E_2$**:** It can be seen that $I_{10} = x \oplus x_2 \oplus x_3$ is statistically independent of $x$. As all the elements in $E_2 \times E_2$ are functions of $I_{10}$, by applying Lemma 5.2, we can infer that $E_2 \times E_2$ is statistically independent of $x$.

3. $E_3 \times E_3$**:** It is also straightforward to see that $E_3 \times E_3$ is statistically independent of $x$ as $x - (A_2 - A_3)$ is statistically independent of $x$.

4. $E_1 \times E_2$**:** $E_1$ is statistically independent of $x_2 \oplus x_3$ and functionally independent of $x$. According to Lemma 5.2, $E_1 \times \{x \oplus x_2 \oplus x_3\}$ is statistically independent

of $x$. Therefore, according to Lemma 5.1, $E_1 \times E_2$ is statistically independent of $x$.

5. $E_1 \times E_3$**:** As $E_1$ is statistically independent of $A_2 - A_3$, $E_1 \times \{x - (A_2 - A_3)\}$(i.e. $E_1 \times E_3$) is also statistically independent of $x$. As the pair $(x \oplus x_2 \oplus x_3, (x - A_2) - A_3)$ is statistically independent of $x$, $(I_{10} \cup I_{11} \cup I_{12}) \times E_3$ is statistically independent of $x$, because all these can be expressed as a function of $(x \oplus x_2 \oplus x_3, (x - A_2) - A_3)$.

6. $E_2 \times E_3$**:** Finally, we need to prove that $I_{13} \times E_3$ is statistically independent of $x$ to establish that $E_2 \times E_3$ is statistically independent of $x$. Suppose that $v_1 = (x - A_2) - A_3$ and $v_2 = (x \oplus x_2 \oplus x_3 \oplus a)$. It is easy to see that $v_1$ and $v_2$ are statistically independent of each other as well as $x$. We can write $I_{13} \times E_3$ as $\{v_1 + v_2 - x\} \times v_1$, which is statistically independent of $x$.

From all this it can be concluded that Theorem 5.1 holds. □

**Theorem 5.2.** *Algorithm 29 is secure against second order DPA.*

*Proof.* Assume that the Boolean function $\delta_0(x) = 0$ only when $x = 0$. So, the $compare_b(x, y)$ can be represented as $\delta_0(x \oplus y) \oplus b$. For Algorithm 29 to be secure, it is important that the function $compare_b$ is implemented in a way which prevents any first-order leakage on $compare(x, y)$. One such method is recalled in Algorithm 3 and we can construct the proof based on this result. All the intermediate variables that are used in Algorithm 29 are shown in Table 5.2. It can be seen that the intermediate variables are almost identical to those in Algorithm 28. Hence, we can prove the security using the same arguments as given in the proof of Theorem 5.1. □

Table 5.3: Intermediate variables used in Algorithm 30

| index | $I_{index}$ |
|---|---|
| 1 | $x_2$ |
| 2 | $x_3$ |
| 3 | $A_2$ |
| 4 | $A_3$ |
| 5 | $r$ |
| 6 | $A_2 - r$ |
| 7 | $A_2 - r + A_3$ |
| 8 | $a$ |
| 9 | $a - A_2 + r - A_3$ |
| 10 | $x - A_2 - A_3$ |
| 11 | $x - A_2 - A_3 + a$ |
| 12 | $(x - A_2 - A_3 + a) \oplus x_2$ |
| 13 | $(x - A_2 - A_3 + a) \oplus x_2 \oplus x_3$ |
| 14 | $x \oplus x_2 \oplus x_3$ |

Table 5.4: Intermediate variables used in Algorithm 31

| index | $I_{index}$ |
|---|---|
| 1 | $x_2$ |
| 2 | $x_3$ |
| 3 | $A_2$ |
| 4 | $A_3$ |
| 5 | $b$ |
| 6 | $a$ |
| 7 | $a - A_2$ |
| 8 | $\delta_0((a - A_2) \oplus A_3) \oplus b$ |
| 9 | $x - A_2 - A_3$ |
| 10 | $x - A_2 - A_3 + a$ |
| 11 | $(x - A_2 - A_3 + a) \oplus x_2$ |
| 12 | $(x - A_2 - A_3 + a) \oplus x_2 \oplus x_3$ |
| 13 | $x \oplus x_2 \oplus x_3$ |

**Theorem 5.3.** *Algorithm 30 is secure against second order DPA.*

*Proof.* We list all the intermediate variables that are used in Algorithm 30 in Table 5.3. We can use similar arguments as in Theorem 5.1 to prove that no pair of the intermediate variables is statistically dependent on $x$. $\square$

**Theorem 5.4.** *Algorithm 31 is secure against second order DPA.*

*Proof.* We list all the intermediate variables that are used in Algorithm 31 in Table 5.4. Again, the proof of security can be constructed similar to that of Theorem 5.1, by proving that no pair of the intermediate variables is statistically dependent on $x$. $\square$

## 5.4 Implementation Results

We implemented all the algorithms from Section 5.1 and Section 5.2 in both Matlab and ANSI C. We considered the simplest case of converting between 8-bit masks. The Matlab implementation served as reference for the C implementation so that we could easily verify the correctness of the latter. We tested each algorithm individually using 100,000 pseudo-random inputs and found that all the algorithms produce the correct result for all the test cases. In our C implementation, the random numbers are generated with the help of the `rand()` function of the standard C library[1]. Though this is sufficient for testing purposes, a real-world implementation would require pseudo-random numbers of better quality. Furthermore, it should be noted that we developed all our implementation mainly for the purpose of having a proof of concept rather than achieving high performance. The implementation can be further optimized, which means the results we present in this section should be seen as upper bounds of the execution time. Furthermore, if the conversions are used in a real-world application, one has to ensure that the compilation process respects the flow of intermediate variables assumed in our security analysis. If this is not the case then it may be necessary to develop an assembly language implementation.

The implementations of Algorithm 28 and Algorithm 30 are straightforward. We create a table of 256 bytes and for each possible value of a $(0 \leq a \leq 255\ )$ store the corresponding entry in one byte. The indexing of the table is done through $a'$ and the correct value of the third share is retrieved by accessing the entry corresponding to $r$. For efficient implementations of Algorithm 29 and Algorithm 31, we need to implement the $compare_b$ function as efficiently as possible. We used the following technique to implement the function. We first create an array of 32 bytes and initialize all the bits to $\bar{b}$. We treat the array as a collection of 256 bits, each initialized to $\bar{b}$. Then, for a random value $r_3$, we set the corresponding bit position in the array to $b$. Now, each call to function $compare_b$ is replaced by a single look-up into the array. For example, $compare_b(x, y)$ is obtained by retrieving the value

[1]On an 8-bit AVR processor, e.g. ATmega128, each call of the `rand()` function takes roughly 1500 cycles when using the `avr-libc` library of the WinAVR tool suite.

at the bit position $(x \oplus r_3) \oplus y$. The index of the byte that contains the bit can be found by a logical right-shift operation. The bit itself can be extracted from the byte via a shift operation too.

Table 5.5: Implementation results on 8, 16 and 32-bit platforms

| Algorithm | Cycles | RAM (bytes) |
|---|---|---|
| 8-bit architecture | | |
| Algorithm 28 | 5769 | 256 |
| Algorithm 29 | 6742 | 32 |
| Algorithm 30 | 5769 | 256 |
| Algorithm 31 | 6742 | 32 |
| 16-bit architecture | | |
| Algorithm 28 | 4983 | 256 |
| Algorithm 29 | 16706 | 32 |
| Algorithm 30 | 4983 | 256 |
| Algorithm 31 | 16706 | 32 |
| 32-bit architecture | | |
| Algorithm 28 | 793 | 256 |
| Algorithm 29 | 1087 | 32 |
| Algorithm 30 | 793 | 256 |
| Algorithm 31 | 1087 | 32 |

In order to evaluate the execution time of the algorithms, we compiled them for the 32-bit ARM platform as well as 8-bit AVR platforms and performed simulations with AVR Studio. We also evaluated the algorithms on a low-power 16-bit micro-controller, namely the TI MSP430, with the help of a cycle-accurate instruction-set simulator. Table 5.5 shows the results of our implementation on these three platforms. In Table 5.5, the second column specifies the amount of time required to convert an 8-bit mask from one form to other in number of clock cycles. The third column gives the memory requirements of the algorithms in number of bytes. As we can see, the algorithms using table look-ups, i.e. Algorithm 28 and Algorithm 30, are faster than the ones which do not use tables. This is because of the additional time required to evaluate the $compare_b$ function in case of Algorithm 29 and Algorithm 31. At the same time, the algorithms using the table computation method require eight times more memory than their counterparts.

Note here that the execution times for Algorithm 29 and Algorithm 31 on the 16-bit platform are somewhat misleading. From Table 5.5, it can be seen that the execution time on the MSP430 is almost 2.5 times slower than that on the 8-bit AVR. This is due to the fact that the used MSP430 processor does not have a barrel shifter, which means a shift operation by $n$ bit positions takes $n$ clock cycles. On the other hand, the AVR features a fast barrel shifter able to execute all shift operations in a single cycle, independent of the shift distance.

## 5.5 Efficient Second-order Secure Boolean to Arithmetic Masking

Though the algorithms given in Section 5.1 require only 3 shares, they become infeasible for implementation on low-resource devices like smart cards for $n > 10$ (the additions are performed modulo $2^n$), as we require lookup table of size $2^n$. In this section we propose new secure conversion algorithms, which overcome the above limitation and can be easily applied to cryptographic constructions with arbitrary $n$, *e.g.* HMAC-SHA-1 with $n = 32$.

### 5.5.1 Improved Conversion Algorithm

The problem here is, we are given three Boolean shares $x_1, x_2, x_3$ so that the sensitive variable $x$ is obtained by $x = x_1 \oplus x_2 \oplus x_3$. The goal is to find three arithmetic shares $A_1, A_2, A_3$ satisfying $x = A_1 + A_2 + A_3$ without leaking any first or second-order information about $x$. This can be achieved by generating two shares $A_2$ and $A_3$ randomly and computing the third share as: $A_1 = x - A_2 - A_3$ as done in [VG13] using the approach followed by Rivain et al. in [RDP08]. But as mentioned earlier, their scheme becomes infeasible to be used in practice when $n > 10$, as it requires a lookup table of size $2^n$. To obtain a solution for $n > 10$, we use divide and conquer approach. That is, we divide each share into $p$ words of $l$ bits each, and compute $(A_1^i)_{(0 \le i \le p-1)}$ independently, where $A_1 = A_1^{p-1} || \cdots || A_1^0$. In this case, we also need to handle the carries from word $i$ to word $i+1$. These carries in turn also need to be protected by masking, which can leak information about the sensitive variable otherwise. In the following, we present our method to protect the sensitive variables along with carries and demonstrate its security with a formal proof.

We differentiate between two sets of carries: input carries i.e., carries used in computing $A_1^i$ and output carries i.e., carries raised while computing $A_1^i$. As computing $A_1^i$ involves two subtractions, there will be two output carries from each word $i$, which become input carries for the word $i+1$. For the first word, input carries are initialized to 0, i.e., $c_1^0 = 0, c_2^0 = 0$. We compute $A_1^i$ from the input $x^i$ and carries $c_1^i, c_2^i$ as follows:

$$A_1^i = (x^i -_l c_1^i -_l A_2^i -_l c_2^i -_l A_3^i)$$

Here the operation $a -_l b$ represents $(a - b) \mod 2^l$. Similarly, the output carries $c_1^{i+1}, c_2^{i+1}$ are computed as follows:

$$c_1^{i+1} = \mathsf{Carry}(x^i, c_1^i) \oplus \mathsf{Carry}(x^i -_l c_1^i, A_2^i) \tag{5.1}$$

$$c_2^{i+1} = \mathsf{Carry}(x^i -_l c_1^i -_l A_2^i, c_2^i) \oplus \mathsf{Carry}(x^i -_l c_1^i -_l A_2^i -_l c_2^i, A_3^i) \tag{5.2}$$

where $\mathsf{Carry}(a, b)$ represents the carry from the operation $(a - b)$. Note that each of the carry computation involves two subtractions: one with the input carry and the other with one of the random shares i.e., $A_2^i$ or $A_3^i$. In the simplest case, a subtraction $a - b$ produces a carry if $a < b$. However, in our case, we have operations of the form $(a -_l c) -_l b$ where both $a$ and $b$ are $l$-bit integers and $c$ is either 0 or 1. In the case of $c = 0$, the above operation generates a carry if $a < b$. But when $c = 1$, we have to consider another case, namely $a < c$, which can only happen if $a = 0$ and $c = 1$. In this special case, the difference $a -_l c$ becomes $2^l - 1$, thereby producing a carry that needs to be handled as well. However, there won't be a carry from the second subtraction as $b \leq 2^{l-1}$. Namely, the carries from these two cases are mutually exclusive; hence the output carry is set to one when either of them produces a carry as shown in (5.1) and (5.2). For simplicity, we define functions $F_1 : \{0,1\}^{l+1} \to \{0,1\}^{l+1}$, $F_2 : \{0,1\}^{2l} \to \{0,1\}^{l+1}$ as follows:

$$F_1(a,b) = a -_l b || (\mathsf{Carry}(a,b)) \tag{5.3}$$

$$F_2(a,b) = a -_l b || (\mathsf{Carry}(a,b)) \tag{5.4}$$

For the word $i$, we can compute $A_1^i$ as well as the output carries $c_1^{i+1}, c_2^{i+1}$ using $F_1$ and $F_2$ as follows:

$$\begin{aligned}(B_1^i || d_1^i) &= F_1(x^i, c_1^i)\\ (B_2^i || d_2^i) &= F_2(B_1^i, A_2^i)\\ (B_3^i || d_3^i) &= F_1(B_2^i, c_2^i)\\ (B_4^i || d_4^i) &= F_2(B_3^i, A_3^i)\end{aligned}$$

where $A_1^i = B_4^i$ and $c_1^{i+1} = d_1^i \oplus d_2^i, c_2^{i+1} = d_3^i \oplus d_4^i$. According to [RDP08], the S-box must be balanced for their scheme to be secure [2]. In our case, the function $F_1$ plays the same role and is balanced. Hence, the security guarantee is preserved. We first present non-randomized version of our solution below for simplicity.

**Algorithm 32** Insecure 2OB→A

```
Input: Sensitive variable: $x = x_1 \oplus x_2 \oplus x_3$
Output: Arithmetic shares: $x = A_1 + A_2 + A_3$
 1: $c_1^0, c_2^0 \leftarrow 0$                                  ▷ Initially carry is zero
 2: for $i := 0$ to $p - 1$ do
 3:     $A_2^i, A_3^i \leftarrow \mathsf{Rand}(l)$                 ▷ Generate output masks randomly
 4:     $(B_1^i, d_1^i) \leftarrow F_1(x^i, c_1^i)$
 5:     $(B_2^i, d_2^i) \leftarrow F_2(B_1^i, A_2^i)$
 6:     $(B_3^i, d_3^i) \leftarrow F_1(B_2^i, c_2^i)$
 7:     $(B_4^i, d_4^i) \leftarrow F_2(B_3^i, A_3^i)$
 8:     $(A_1^i, c_1^{i+1}, c_2^{i+1}) \leftarrow (B_4^i, d_1^i \oplus d_2^i, d_3^i \oplus d_4^i)$
 9: end for
10: return $A_1, A_2, A_3$
```

The challenge now is to obtain the same result without leaking any first or second-order information about the sensitive variable $x$ as well as the carries $c_1^i, c_2^i$ for $0 \leq i \leq p-1$. We present our solution in two parts: we fist give the algorithm to

[2] An S-box $S : \{0,1\}^n \to \{0,1\}^m$ is said to be balanced if every element in $\{0,1\}^m$ is image of exactly $2^{n-m}$ elements in $\{0,1\}^n$ under $S$.

compute the result for one word i.e. $A_1^i$ securely; then we use this as a subroutine to compute $A_1$. Our solution given in Algorithm 33 uses the similar technique used by Rivain et al in [RDP08] (Recalled in Algorithm 28) in combination with Algorithm 32. Algorithm 33 takes as input: three Boolean shares, six input carry shares (three each for the two carries), two output arithmetic shares and four output carry shares. It returns the third arithmetic share and the remaining two output carry shares. Similar to Algorithm 28, we create a lookup table $T$ for all the possible values in $[0, 2^{l+2} - 1]$. Here $l$ bits are used for storing $A_1^i$ and two bits for the two carries correspondingly. As we can see, the rest of the algorithm is similar to the original algorithm except for handling two extra bits for the carry. [3]

**Algorithm 33** Sec2OB→A_Word

```
Input: Three input shares: $(x_1^i = x^i \oplus x_2^i \oplus x_3^i, x_2^i, x_3^i) \in \mathbb{F}_{2^l}$, Six input carry shares:
    $g_1^i = c_1^i \oplus g_2^i \oplus g_3^i, g_2^i, g_3^i$, $g_4^i = c_2^i \oplus g_5^i \oplus g_6^i, g_5^i, g_6^i \in \mathbb{F}_2$, Output arithmetic shares: $A_2^i, A_3^i$,
    Output carry shares: $h_1^i, h_2^i, h_3^i, h_4^i$
Output: Masked Arithmetic share: $(x^i -_l A_2^i) -_l A_3^i$ and masked output carries
 1: $r_1 \leftarrow \mathsf{Rand}(l)$; $r_2 \leftarrow \mathsf{Rand}(1)$; $r_3 \leftarrow \mathsf{Rand}(1)$
 2: $r_1' \leftarrow (r_1 \oplus x_2^i) \oplus x_3^i$; $r_2' \leftarrow (r_2 \oplus g_2^i) \oplus g_3^i$; $r_3' \leftarrow (r_3 \oplus g_5^i) \oplus g_6^i$;
 3: for $a_1 := 0$ to $2^l - 1$, $a_2 := 0$ to 1, $a_3 := 0$ to 1 do
 4:     $a_1' \leftarrow a_1 \oplus r_1'$; $a_2' \leftarrow a_2 \oplus r_2'$; $a_3' \leftarrow a_3 \oplus r_3'$
 5:     $(B_1^i, d_1^i) \leftarrow F_1((x_1^i \oplus a_1), (g_1^i \oplus a_2))$
 6:     $(B_2^i, d_2^i) \leftarrow F_2(B_1^i, A_2^i)$
 7:     $(B_3^i, d_3^i) \leftarrow F_1(B_2^i, (g_4^i \oplus a_3))$
 8:     $(B_4^i, d_4^i) \leftarrow F_2(B_3^i, A_3^i)$
 9:     $e_1^i \leftarrow ((d_1^i \oplus h_1^i) \oplus d_2^i) \oplus h_2^i$
10:     $e_2^i \leftarrow ((d_3^i \oplus h_3^i) \oplus d_4^i) \oplus h_4^i$
11:     $(T_1[a_1'||a_2'||a_3'], T_2[a_1'||a_2'||a_3'], T_3[a_1'||a_2'||a_3']) \leftarrow (B_4^i, e_1^i, e_2^i)$
12: end for
13: return $T_1[r_1||r_2||r_3], T_2[r_1||r_2||r_3], T_3[r_1||r_2||r_3]$
```

Finally, we give our second-order secure method to obtain three arithmetic shares corresponding to the three Boolean shares in Algorithm 34. For the first word (i.e. $i = 0$), there are no input carries. Hence, the three shares for both the carries are set to zero (Step 1). Here, $g_1^0 = g_2^0 = g_3^0 = c_1^0 = 0$ and $g_4^0 = g_5^0 = g_6^0 = c_2^0 = 0$. To protect the output carries, we use four uniformly generated random bits: $h_1^i, h_2^i, h_3^i, h_4^i$; two each for the two carries. The third share for the carries as well as $A_1^i$ are computed recursively using the function Sec2OB→A_Word (Algorithm 33) [4]. Note here that for word $i$, $g_1^i \oplus g_2^i \oplus g_3^i = c_1^i$ and $g_4^i \oplus g_5^i \oplus g_6^i = c_2^i$. The time complexity of the overall solution is $\mathcal{O}(2^{l+2} \cdot p)$ and the memory required is $(2^{l+2} \cdot (l+2))$ bits.

[3]We use different tables for storing the value and the carries so that the security proof can be easily obtained as in [RDP08].

[4]Every call to the function Sec2OB→A_Word creates a new table and used for that particular word only. Hence unlike the original method in [RDP08], we don't reuse the table.

**Algorithm 34** EffSec2OB→A

```
Input: Boolean shares: $x_1 = x \oplus x_2 \oplus x_3, x_2, x_3$
Output: Arithmetic shares: $A_1, A_2, A_3$ so that $x = A_1 + A_2 + A_3$
1: $g_1^0, g_2^0, g_3^0, g_4^0, g_5^0, g_6^0 \leftarrow 0$          ▷ Initially carry is zero
2: for $i := 0$ to $p - 1$ do
3:     $A_2^i, A_3^i \leftarrow \mathsf{Rand}(l)$          ▷ Generate output masks randomly
4:     $h_1^i, h_2^i, h_3^i, h_4^i \leftarrow \mathsf{Rand}(1)$
5:     $(A_1^i, g_1^{i+1}, g_4^{i+1}) \leftarrow \mathsf{Sec2OB{\to}A\_Word}\ ((x_j^i)_{1\le j\le 3}, (g_j^i)_{1\le j\le 6}, A_2^i, A_3^i, (h_j^i)_{1\le j\le 4})$
6:     $g_2^{i+1}, g_3^{i+1}, g_5^{i+1}, g_6^{i+1} \leftarrow h_1^i, h_2^i, h_3^i, h_4^i$
7: end for
8: return $A_1, A_2, A_3$
```

### 5.5.2 Security Analysis.

For an algorithm to be secure against second-order attacks, no pair of the intermediate variables appearing in the algorithm should jointly leak the sensitive variable. In [RDP08] the authors prove the security by enumerating all the possible pairs of intermediate variables and showing that the joint distribution of none of these pairs is dependent on the distribution of the sensitive variable. We use similar method to prove the security of Algorithm 33. We then prove the security of Algorithm 34 using induction.

**Lemma 5.3.** *Algorithm 33 is secure against second-order DPA.*

*Proof.* We list all the intermediate variables used in Algorithm 28 and Algorithm 33 in Table 5.6. The intermediate variables computed using similar technique appear in the same row. The only difference is that we have three intermediate variables instead of one for each row. [5] Hence, the security of Algorithm 33 can be derived from the same arguments as in case of Algorithm 28.

□

**Theorem 5.5.** *Algorithm 34 is secure against second-order DPA.*

*Proof.* To prove the security of Algorithm 34, we apply mathematical induction on the number of words $p$. When $p = 1$, we already know that the algorithm is secure from Lemma 5.3. Now assume that the algorithm is secure for $p = n$. Let $E_i$ be the set that represents the collection of all the intermediate variables corresponding to the word $i$. Then, according to the induction hypothesis, $\{E_1, \cdots E_n\} \times \{E_1, \cdots E_n\}$ is independent of the sensitive variables $x$, $c_1$ and $c_2$.

For the algorithm to be secure when $p = n + 1$, the set $\{E_1, \cdots E_n, E_{n+1}\} \times \{E_1, \cdots E_n, E_{n+1}\}$ should be independent of the sensitive variables $x$, $c_1$ and $c_2$. Without loss of generality, we divide this set into three subsets: $\{E_{n+1} \times E_{n+1}\}$, $\{E_1, \cdots E_n\} \times \{E_1, \cdots E_n\}$, $\{E_{n+1}\} \times \{E_1, \cdots E_n\}$. The security of $\{E_{n+1} \times E_{n+1}\}$ can be established directly from the base case and the security of $\{E_1, \cdots E_n\} \times \{E_1, \cdots E_n\}$ follows from the induction hypothesis. All the intermediate variables in $E_{n+1}$ fall into two categories: the variables that are generated randomly and

[5]The only exception is for the row $S(x_1 \oplus a)$, where we have four variables.

| **Intermediate variables in Algorithm 28** | **Intermediate variables in Algorithm 33** |
|---|---|
| $x_2$ | $x_2^i, g_2^i, g_5^i$ |
| $x_3$ | $x_3^i, g_3^i, g_6^i$ |
| $y_1$ | $A_2^i, h_1^i, h_3^i$ |
| $y_2$ | $A_3^i, h_2^i, h_4^i$ |
| $r$ | $r_1, r_2, r_3$ |
| $x_2 \oplus r$ | $x_2^i \oplus r_1, g_2^i \oplus r_2, g_5^i \oplus r_3$ |
| $x_2 \oplus r \oplus x_3$ | $x_2^i \oplus r_1 \oplus x_3^i, g_2^i \oplus r_2 \oplus g_3^i, g_5^i \oplus r_3 \oplus g_5^i$ |
| $a$ | $a_1, a_2, a_3$ |
| $a \oplus r \oplus x_2 \oplus x_3$ | $a_1 \oplus r'_1, a_2 \oplus r'_2, a_3 \oplus r'_3$ |
| $x_1 = x \oplus x_2 \oplus x_3$ | $x_1^i = x^i \oplus x_2^i \oplus x_3^i, g_1^i = c_1^i \oplus g_2^i \oplus g_3^i, g_4^i = c_2^i \oplus g_3^i \oplus g_6^i$ |
| $x_1 \oplus a$ | $x_1^i \oplus a, g_1^i \oplus a_2, g_4^i \oplus a_3$ |
| $S(x_1 \oplus a)$ | $(B_1^i \Vert d_1^i) = F_1((x_1^i \oplus a), g_1^i \oplus a_2)$ <br> $(B_3^i \Vert d_3^i) = F_1((x_1^i \oplus a) -_l g_1^i \oplus a_2 -_l A_2^i, g_4^i \oplus a_3)$ <br> $d_2^i = \mathsf{Carry}((x_1^i \oplus a) -_l (g_1^i \oplus a_2), A_2^i)$ <br> $d_4^i = \mathsf{Carry}((x_1^i \oplus a) -_l (g_1^i \oplus a_2) -_l A_2^i -_l (g_4^i \oplus a_3), A_3^i)$ |
| $S(x_1 \oplus a) \oplus y_1$ | $B_2^i = (x_1^i \oplus a) -_l (g_1^i \oplus a_2) -_l A_2^i,$ <br> $d_1^i \oplus h_1^i \oplus d_2^i, d_3^i \oplus h_3^i \oplus d_4^i$ |
| $S(x_1 \oplus a) \oplus y_1 \oplus y_2$ | $B_4^i = (x_1^i \oplus a) -_l (g_1^i \oplus a_2) -_l A_2^i -_l (g_4^i \oplus a_3) -_l A_3^i,$ <br> $d_1^i \oplus h_1^i \oplus d_2^i \oplus h_2^i, d_3^i \oplus h_3^i \oplus d_4^i \oplus h_4^i$ |
| $S(x) \oplus y_1 \oplus y_2$ | $x^i -_l c_1^i -_l A_2^i -_l c_2^i -_l A_3^i,$ <br> $c_1^{i+1} \oplus h_1^i \oplus h_2^i, c_2^{i+1} \oplus h_3^i \oplus h_4^i$ |

Table 5.6: Comparison between Intermediate variables used in Algorithm 28 and Algorithm 33

are independent of any variables in $\{E_1, \cdots E_n\}$; and the variables which are a function of one or more of the following: $(x^{n+1})$, $(c_1^{n+1})$ and $(c_2^{n+1})$. Any pair of the intermediate variables involving the first category are independent of the sensitive variables by definition and the first-order resistance of $\{E_1, \cdots E_n\}$. The carry shares for the word $n+1$: $(c_i^{n+1})_{1 \le i \le 2}$ are computed from the word $n$. Hence the security of $(c_i^{n+1})_{1 \le i \le 3} \times \{E_1, \cdots E_n\}$ is already established in $\{E_n\} \times \{E_1, \cdots E_n\}$. It is easy to see that the set $(x^{n+1}) \times \{E_1, \cdots E_n\}$ is independent of any sensitive variable. Hence, the set $\{E_{n+1}\} \times \{E_1, \cdots E_n\}$ is also independent of any sensitive variable, which proves the theorem. □

## 5.6 Efficient Second-order Secure Arithmetic to Boolean Masking

In arithmetic to Boolean conversion, the problem is to find three shares $x_1, x_2, x_3$ satisfying $x = x_1 \oplus x_2 \oplus x_3$, where the sensitive variable $x$ is represented by three arithmetic shares $A_1, A_2, A_3$ with $x = A_1 + A_2 + A_3$. To solve this problem, we follow the same strategy as in Section 5.5.1. We generate two Boolean shares $x_2$ and $x_3$ randomly, and compute the third share by using the relation $x_1 = ((A_1 + A_2 + A_3) \oplus x_2 \oplus x_3)$, without leaking the value of $x$ to first or second-order DPA. We use the following approach: we first obtain a method to convert a single arithmetic share word; then we apply this procedure recursively to all the words. For each word, we have to deal with two carries corresponding to the two additions, i.e., the carry from the addition of the shares corresponding to $A_2, A_3$

and its subsequent addition with $A_1$.

**Algorithm 35** Sec20A→B_Word

```
Input: Three input shares: $(A_1^i = (x^i - A_2^i) - A_3^i, A_2^i, A_3^i) \in \mathbb{F}_{2^l}$, Six input carry shares:
    $g_1^i = c_1^i \oplus g_2^i \oplus g_3^i, g_2^i, g_3^i$, $g_4^i = c_2^i \oplus g_5^i \oplus g_6^i, g_5^i, g_6^i \in \mathbb{F}_2$, Output Boolean shares: $x_2^i, x_3^i$,
    Output carry shares: $h_1^i, h_2^i, h_3^i, h_4^i$
Output: Third Boolean share: $x_1^i = x^i \oplus x_2^i \oplus x_3^i$ and masked output carries
 1: $r_1 \leftarrow \mathsf{Rand}(l)$; $r_2 \leftarrow \mathsf{Rand}(1)$; $r_3 \leftarrow \mathsf{Rand}(1)$
 2: $r_1' \leftarrow (A_2^i - r_1) + A_3^i$                      ▷ Mask two arithmetic shares
 3: $r_2' \leftarrow (r_2 \oplus g_2^i) \oplus g_3^i$; $r_3' \leftarrow (r_3 \oplus g_5^i) \oplus g_6^i$
 4: for $a_1 := 0$ to $2^l - 1$ do
 5:     $a_1' \leftarrow a_1 -_l r_1'$          ▷ $a_1' = r_1 \implies a_1 = r_1 +_l ((A_2^i - r_1) + A_3^i)$
 6:     for $a_2 := 0$ to 1, $a_3 := 0$ to 1 do
 7:         $a_2' \leftarrow a_2 \oplus r_2'$; $a_3' \leftarrow a_3 \oplus r_3'$
 8:         $(B_1^i || d_2^i) \leftarrow F_3(A_1^i + (a_3 \oplus g_4^i) + ((a_2 \oplus g_1^i) +_l a_1))$
 9:         $d_1^i \leftarrow \mathsf{Carry}(a_1, r_1') \oplus \mathsf{Carry}(a_1, -(a_2 \oplus g_1^i))$
10:         $x_1^i \leftarrow (B_1^i \oplus x_2^i) \oplus x_3^i$        ▷ Apply Boolean masking to the result
11:         $e_1^i \leftarrow (d_1^i \oplus h_1^i) \oplus h_2^i$            ▷ Apply masking to the carries
12:         $e_2^i \leftarrow (d_2^i \oplus h_3^i) \oplus h_4^i$
13:         $T_1[a_1'||a_2'||a_3'], T_2[a_1'||a_2'||a_3'], T_3[a_1'||a_2'||a_3'] \leftarrow (x_1^i, e_1^i, e_2^i)$
14:     end for
15: end for
16: return $T_1[r_1||r_2||r_3], T_2[r_1||r_2||r_3], T_3[r_1||r_2||r_3]$
```

Algorithm 35 gives the solution for converting one word of Boolean shares to corresponding arithmetic shares. We again use the technique from Algorithm 28 as in Algorithm 33. As the input shares here are masked using arithmetic masking instead of Boolean masking, we have to modify the operations accordingly. Hence, the computation of $r_1'$ (Step 2) and $a_1'$ (Step 5) are replaced with additive operations. However, we can still mask the carries using Boolean masking as previously and hence the corresponding operations do not change (Step 3, Step 7). We create a table for all possible values in $[0, 2^{l+2} - 1]$, where $l$ bits are used for $x_1^i$ and the extra two bits for the carries. From $a_1' = a_1 -_l r_1'$, we have $a_1 = a_1' +_l r_1'$. However, $a_1 - r_1'$ could generate a carry, which needs to be taken care while computing $x_1^i$. Hence, we add the previous carry $(a_2 \oplus g_2^i)$ to $a_1$ to get $A_2^i +_l A_3^i +_l c_1^i$ as follows:

$$a_1 +_l (a_2 \oplus g_1^i) = (r_1 +_l ((A_2^i - r_1) + A_3^i) +_l c_1^i) = A_2^i +_l A_3^i +_l c_1^i$$

when $a_1' = r_1$ and $a_2' = r_2$. The out carry $d_1^i$ (which becomes $c_1^{i+1}$ for the next word) can occur in two scenarios: when $a_1 < r_1'$ or when $(a_1 + (a_2 \oplus g_1^i)) \geq 2^l$ (Step 9). It is easy to see that these two cases are mutually exclusive. Now to compute $x_1^i$, we use function $F_3 : \{0,1\}^{l+1} \to \{0,1\}^{l+1}$, which is defined as:

$$F_3(a) = a \mod 2^l || \mathsf{Carry}(2^l, a)$$

We then call $F_3$ with $(A_1^i + (a_3 \oplus g_4^i) + ((a_2 \oplus g_1^i) +_l a_1))$ where $a_3$ represents two shares of the second carry. In this case, the first part returned by $F_3$ gives $x^i$, and the second part corresponds to the second carry which becomes $c_2^{i+1}$ for the next word [6]. Namely, when $a'_1 = r_1$, $a'_2 = r_2$ and $a'_3 = r_3$ we have:

$$F_3(A_1^i + (a_3 \oplus g_4^i) + ((a_2 \oplus g_1^i) +_l a_1)) = (x^i + (a_3 \oplus g_4^i)) \bmod 2^l || \\ \mathsf{Carry}(2^l, (x^i + (a_3 \oplus g_4^i)))$$

Once we have $x^i$ and the carries $d_1^i, d_2^i$, we can simply apply Boolean masks on them to obtain $x_1^i$ and the masked carries (Steps 10, 11 and 12).

Finally we give the full algorithm to convert from arithmetic to Boolean masking in Algorithm 36. It is similar to Algorithm 34 except that the Boolean shares and arithmetic shares are interchanged.

**Algorithm 36** EffSec20A→B

```
Input: Arithmetic shares: $A_1 = x - A_2 - A_3, A_2, A_3$
Output: Boolean shares: $x_1, x_2, x_3$ so that $x = x_1 \oplus x_2 \oplus x_3$
1: $g_1^0, g_2^0, g_3^0, g_4^0, g_5^0, g_6^0 \leftarrow 0$                ▷ Initially carry is zero
2: for $i := 0$ to $p - 1$ do
3:     $x_2^i, x_3^i \leftarrow \mathsf{Rand}(l)$                ▷ Generate output masks randomly
4:     $h_1^i, h_2^i, h_3^i, h_4^i \leftarrow \mathsf{Rand}(1)$
5:     $(x_1^i, g_1^{i+1}, g_4^{i+1}) \leftarrow$ Sec20A→B_Word $((A_j^i)_{1\le j\le 3}, (g_j^i)_{1\le j\le 6}, x_2^i, x_3^i, (h_j^i)_{1\le j\le 4})$
6:     $g_2^{i+1}, g_3^{i+1}, g_5^{i+1}, g_6^{i+1} \leftarrow h_1^i, h_2^i, h_3^i, h_4^i$
7: end for
8: return $x_1, x_2, x_3$
```

**Theorem 5.6.** *Algorithm 36 is secure against second-order DPA.*

*Proof.* The proof of Algorithm 36 can be obtained similar to Algorithm 34 and is omitted. □

## 5.7 First-order Secure Masked Addition with Lookup Tables

As already established, this paper focuses on the problem of dealing with arithmetic operations on Boolean masks. Till now, we solved this problem by converting the Boolean masks to arithmetic masks. The idea is that once we have the arithmetic masks, we can perform arithmetic operations directly and then convert the result back to Boolean masks. But there also exist an alternative solution to the original problem i.e., devising a solution to perform addition directly on the Boolean masks. This idea was first studied with respect to first-order masking in [BN05] and precised in [KRJ04]. In this section, we provide an alternative method using lookup tables based on the conversion method proposed by Debraize [Deb12].

The problem here is: we are given Boolean shares of two $n$-bit sensitive variables $x : x_1, r$ and $y : y_1, s$. We need to compute $z_1$ so that $z_1 \oplus r \oplus s = x + y$, without any

[6]Note here that even though $x^i$ and the carries are computed in clear, they are hidden among $2^{l+2} - 1$ dummy computations, which is the main basis for Rivain et al's original algorithm.

first-order leakage of $x$ and $y$. We follow the similar divide and conquer approach used in Section 5.5 and Section 5.6. Namely, we divide $n$-bit shares into $p$ words of $l$-bit each and perform addition on the words independently. Moreover, our method also masks the carry from word $i$ to word $i+1$. The addition of each word is performed using a lookup table, which can be reused for all the words. [7]

Our algorithm to generate the lookup table is given in Algorithm 37. It creates a table of $2^{2l+1}$ entries, where each entry requires $l+1$ bit memory. Here, $2l$ bits are used for two $l$ -bit inputs $x^i$, $y^i$ and one bit for the input carry. The output consists of $l$-bit $z^i$ and one bit carry. We run through all the possible $2^{2l+1}$ values and store the masked value of sum and carry in the lookup table. Note here that the inputs masks are $t_1, t_2$ and $\rho$ (carry); the out masks are $t_1$ and $\rho$ (carry).

**Algorithm 37** GenTable

```
Input:
Output: Table $T$, $t_1$, $t_2$, $\rho$
1: $t_1, t_2 \leftarrow \mathsf{Rand}(l)$; $\rho \leftarrow \mathsf{Rand}(1)$
2: for $A = 0$ to $2^l - 1$ do
3:     for $B = 0$ to $2^l - 1$ do
4:         $T[\rho||A||B] \leftarrow ((A \oplus t_1) + (B \oplus t_2)) \oplus (\rho||t_1)$
5:         $T[\rho \oplus 1||A||B] \leftarrow ((A \oplus t_1) + (B \oplus t_2) + 1) \oplus (\rho||t_1)$
6:     end for
7: end for
8: return $T, t_1, t_2, \rho$
```

The full algorithm to compute addition on Boolean shares is given in Algorithm 38. Initially, the carry is zero which is masked with the carry mask $\rho$ from Algorithm 37. We differentiate between carry and no carry cases as follows: if $\beta = \rho$ then there is no carry; otherwise, $\beta = \rho \oplus 1$. Before accessing the lookup table, we change the input masks to $t_1$ and $t_2$ (step 3, 4). After we obtain the masked sum, we change the mask back to $r^i \oplus s^i$ from $t_1$ (step 6). Finally the output can be obtained as $z_1 = z_1^{p-1}|| \cdots ||z_1^0 = (x+y) \oplus r \oplus s$.

**Algorithm 38** Sec10A

```
Input: $x_1 = x \oplus r, r, y_1 = y \oplus s, s, T, t_1, t_2, \rho$
Output: $z_1 = (x+y) \oplus r \oplus s$
1: $\beta \leftarrow \rho$
2: for $i = 0$ to $p-1$ do
3:     $x_1^i \leftarrow x_1^i \oplus t_1 \oplus r^i$
4:     $y_1^i \leftarrow y_1^i \oplus t_2 \oplus s^i$
5:     $(\beta||z_1^i) \leftarrow T[\beta||x_1^i||y_1^i]$
6:     $z_1^i \leftarrow (z_1^i \oplus r^i \oplus s^i) \oplus (t_1)$
7: end for
8: return $z_1$
```

**Lemma 5.4.** *Algorithm 38 is secure against first-order DPA.*

[7] In case of second-order masking, we use different tables. But we can reuse the table for first-order masking.

*Proof.* It is easy to see that the distribution of all the intermediate variables in Algorithm 38 is independent of the sensitive variables $x$ and $y$. Hence the proof is straightforward. □

## 5.8 Implementation Results of Improved Algorithms

| Algorithm | $l$ | Time | Memory | rand |
|---|---|---|---|---|
| second-order conversion | | | | |
| Algorithm 34 | 1 | 12186 | 8 | 226 |
| Algorithm 34 | 2 | 11030 | 16 | 114 |
| Algorithm 34 | 4 | 19244 | 64 | 58 |
| Algorithm 36 | 1 | 10557 | 8 | 226 |
| Algorithm 36 | 2 | 9059 | 16 | 114 |
| Algorithm 36 | 4 | 15370 | 64 | 58 |
| CGV $A \to B$ [CGV14] | - | 54060 | - | 484 |
| CGV $B \to A$ [CGV14] | - | 81005 | - | 822 |
| first-order addition | | | | |
| KRJ addition [KRJ04] | - | 371 | - | 1 |
| Algorithm 38 | 4 | 294 | 512 | 3 |

Table 5.7: Implementation results for $n = 32$ on a 32-bit microcontroller. The column Time denotes the running time in number of clock cycles, rand gives the number of calls to the random number generator function, column $l$ and Memory refers to the word size and memory required in bytes for the table based algorithms.

We implemented all the proposed algorithms on a 32-bit ARM microcontroller. The results are summarized in Table 5.7. We used three different word sizes ($l = 1, 2, 4$) for second-order conversion algorithms and word size $l = 4$ for first-order masked addition.[8] To compare our results with the existing techniques, we also implemented CGV method [CGV14] for second-order conversion and KRJ method [KRJ04] for first-order secure addition. As expected, the improvement in case of second-order conversion algorithms is significant due to the decrease in the number of shares from five to three. From Figure 5.1 we can see that the conversion algorithms give best results for $l = 2$. Our Boolean to arithmetic conversion algorithm with negligible memory requirements (around 8 to 64 bytes) is roughly 86% faster than the CGV algorithm. Similarly, our arithmetic to Boolean conversion algorithm improve the running time by 83%, with equivalent memory requirements [9]. On the other hand, we improve the performance of first-order algorithms by roughly 20%.

To study the implications of our new algorithms on practical implementations, we applied these techniques to HMAC-SHA-1. The corresponding results are summarized in Table 5.8. From Figure 5.2 we can see that in the best case scenario (i.e., $l = 2$), our new algorithms perform 85% better than the existing algorithms. In case

[8]We observed that for $l < 4$ *KRJ* algorithm perform better than ours.

[9]Note that this solution also performs better than the improved conversion using the secure addition from Section 4.4

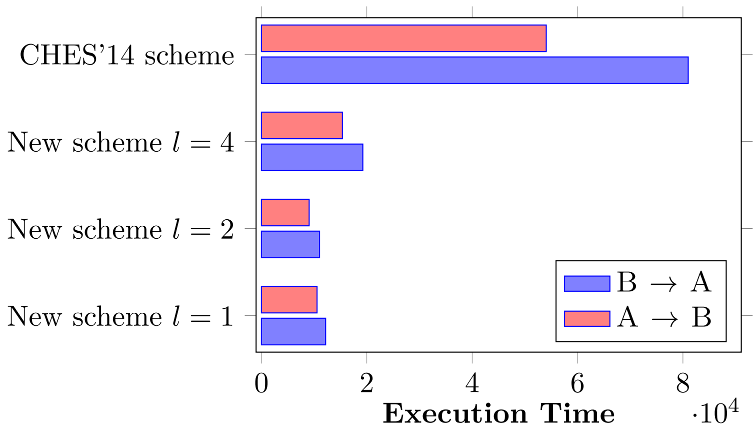


Figure 5.1: Comparison of execution times for $n = 32$ on a 32-bit microcontroller.

| Algorithm | $l$ | Time | PF |
|---|---|---|---|
| HMAC-SHA-1 | - | 104 | 1 |
| second-order conversion | | | |
| Algorithm 34, 36 | 1 | 9715 | 95 |
| Algorithm 34, 36 | 2 | 8917 | 85 |
| Algorithm 34, 36 | 4 | 15329 | 147 |
| CGV [CGV14] | - | 62051 | 596 |
| first-order addition | | | |
| KRJ addition [KRJ04] | - | 328 | 3.1 |
| Algorithm 38 | 4 | 308 | 2.9 |

Table 5.8: Running time in thousands of clock cycles and penalty factor compared to the unmasked HMAC-SHA-1 implementation

of first-order masking, the improvement is around 6% including the precomputation time required to create the table.

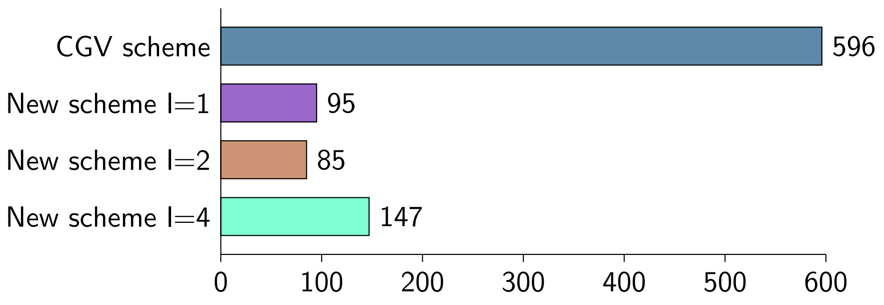


Figure 5.2: Comparison of penalty factor for HMAC-SHA-1 on a 32-bit microcontroller.

# Chapter 6

# Improved Higher-order Secure Masking of AES S-Box

This chapter studies the fast and provably secure higher-order masking of AES S-box proposed by Kim *et al.* at CHES 2011 [KHL11]. Their scheme was mainly an improvement of the higher-order masking scheme for AES proposed by Prouff and Rivain [RP10] using composite field arithmetic. However, the Prouff-Rivain scheme was later found to be insecure by Coron *et al.* [CPRR13]. Based on the same approach used in [CPRR13], we show that the scheme proposed by Kim *et al.* is also insecure. We then repair their scheme using the techniques from [CPRR13]. Our implementation results show that the new scheme gives the best results for higher-order masking of AES. This is a joint work with Junwei Wang, Johann Großschädl and Qiuliang Xu. A part of this work appeared in the proceedings of CT-RSA 2015 [WVGX15].

**Contents**

## 6.1 Introduction

In this section, we recall the two existing schemes for secure higher order masking of AES. We first review the higher-order masking scheme proposed Prouff and Rivain [RP10]. Then we describe the improved scheme proposed by Kim *et al.* [KHL11].

### 6.1.1 Provably Secure Higher-order masking of AES

The first solution to secure AES against higher-order attacks was proposed by Prouff and Rivain at CHES 2010 [RP10]. Their solution was based on the higher-order masking scheme proposed by Ishai, Sahai and Wagner (ISW) to protect any circuit against $t$-limited adversary, who can tap any $t$ wires in the circuit at a given time [ISW03]. The main idea of ISW scheme is to represent the circuit which performs the cryptographic operations as a combination of Boolean gates AND and NOT (which is possible since NAND is a universal gate), and protect these gates independently. Securing NOT gate is easy since $NOT(x_1 \oplus \cdots \oplus x_d) = NOT(x_1) \oplus \cdots \oplus x_d$. To protect AND gate, they proposed an elegant solution where the inputs are elements in $\mathbb{F}_2$. However, Prouff and Rivain later shown that this method can actually be extended to secure multiplication over any field of characteristic 2: $\mathbb{F}_{2^n}$ (as a result also to AES field: $\mathbb{F}_{2^8}$). We recall their solution in Algorithm 39. They also reduced the number of shares required for $d$-th order secure masking scheme to $d+1$ from $2d+1$ required for ISW method.

**Algorithm 39** SecMult

```
Input: $(x_i)$ and $(y_i)$ for $0 \le i \le d$
Output: $(z_i)$ for $0 \le i \le d$, with $\bigoplus_{i=0}^{d} z_i = x \odot y$
 1: for $i = 0$ to $d$ do
 2:     for $j = i + 1$ to $d$ do
 3:         $r_{i,j} \leftarrow \mathsf{rand}(2^n)$                ▷ Generate $n$-bit random
 4:         $r_{j,i} \leftarrow (r_{i,j} \oplus (x_i \odot y_j)) \oplus (x_j \odot y_i)$
 5:     end for
 6: end for
 7: for $i = 0$ to $d$ do
 8:     $z_i = x_i \odot y_i$
 9:     for $j = 1$ to $d$ do
10:         if $i \neq j$ then
11:             $z_i \leftarrow z_i \oplus r_{i,j}$
12:         end if
13:     end for
14: end for
15: return $(z_i)_{0 \le i \le d}$
```

In general, masking a linear function $f$ is easy since for $x = x_1 \oplus \cdots \oplus x_n$ we have:

$$f(x) = f(x_1) \oplus f(x_2) \oplus \cdots \oplus f(x_n) \tag{6.1}$$

All the operations in AES are linear except for the S-box and hence can be easily masked using (6.1). The AES S-box consists of an affine transformation over $\mathbb{F}_{2^8}$ and finding the inverse of an element over the finite field $\mathbb{F}_{2^8}$. As the affine transformation can again be masked directly by modifying (6.1) appropriately, the solution to securely mask AES is simplified to computing the inverse of a filed element securely. Since the inversion $x^{-1}$ of an element $x$ in the finite field $\mathbb{F}_{2^8}$ equals $x^{254}$ over $\mathbb{F}_{2^8} = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$, the authors perform the secure computation of inversion through secure exponentiation, which comprises several secure field multiplications and squarings. We know that in a field of characteristic

2, the squaring operation is linear and hence can be directly masked using (6.1). To securely compute the multiplication, the authors use the algorithm recalled in Algorithm 39.

From the above analysis it is straightforward to see that the main contributor to the computation time is the multiplications, which require $\mathcal{O}(d^2)$ time. Hence, the optimal solution to compute $x^{254}$ should require the least number of multiplications. It is easy to see that we need at least 4 multiplications to compute $x^{254}$ as shown below. Here $S$ denotes the squaring operation and $M$ denotes the multiplication.

$$\begin{tikzcd}
x \arrow[r, "x^2", "S"'] & x^2 \arrow[r, "x^2x", "M"'] & x^3 \arrow[r, "(x^3)^4", "2S"'] & x^{12} \arrow[r, "x^{12}x^3", "M"'] & x^{15} \arrow[r, "(x^{15})^{16}", "4S"'] & x^{240} \arrow[r, "x^{240}x^{12}", "M"'] & x^{252} \arrow[r, "(x^{252})^2", "S"'] & x^{254}
\end{tikzcd}$$

According to [RP10] their scheme is only secure when the inputs $(x_i)_{1\leq i\leq d}$, $(y_i)_{1\leq i\leq d}$ to Algorithm 39 are $d$- independent of each other. Since the inputs used in the above multiplication are not independent, they propose to use RefreshMasks procedure which re- randomizes one of the inputs. Putting altogether, the higher-order secure masking of power function $x^{254}$ can be computed using Algorithm 40.

**Algorithm 40** SecExp254

```
Input: $(x_i)$ for $0 \leq i \leq d$ with $\bigoplus_{i=0}^{d} x_i = x$
Output: $(y_i)$ for $0 \leq i \leq d$ with $\bigoplus_{i=0}^{d} y_i = x^{254}$
1: for $i = 0$ to $d$ do
2:     $z_i \leftarrow x_i^2$
3: end for
4: $(z_i)_{0\leq i\leq d} \leftarrow$ RefreshMasks $((z_i)_{0\leq i\leq d})$
5: $(y_i)_{0\leq i\leq d} \leftarrow$ SecMult$((z_i)_{0\leq i\leq d}, (x_i)_{0\leq i\leq d})$
6: for $i = 0$ to $d$ do
7:     $w_i \leftarrow y_i^4$
8: end for
9: $(w_i)_{0\leq i\leq d} \leftarrow$ RefreshMasks $((w_i)_{0\leq i\leq d})$
10: $(y_i)_{0\leq i\leq d} \leftarrow$ SecMult$((y_i)_{0\leq i\leq d}, (w_i)_{0\leq i\leq d})$
11: for $i = 0$ to $d$ do
12:     $y_i \leftarrow y_i^{16}$
13: end for
14: $(y_i)_{0\leq i\leq d} \leftarrow$ SecMult$((y_i)_{0\leq i\leq d}, (w_i)_{0\leq i\leq d})$
15: $(y_i)_{0\leq i\leq d} \leftarrow$ SecMult$((y_i)_{0\leq i\leq d}, (z_i)_{0\leq i\leq d})$
16: return $(y_i)_{0\leq i\leq d}$
```

Algorithm 40 uses RefreshMasks procedure before performing SecMult on dependent inputs.[1] The RefreshMasks procedure basically modifies the shares using

[1]The function SecMult is only secure when the inputs are $d$-independent of each other. Namely,

freshly generated randoms. Let $(r_i)_{1\leq i\leq d}$ be the new random numbers. Then a call to RefreshMasks $((x_i)_{1\leq i\leq d+1})$ performs the following operation:

$$x_0 = x_0 \oplus \bigoplus_{1\leq i\leq d} r_i$$
$$(x_i)_{1\leq i\leq d} = x_i \oplus r_i$$

We recall the full algorithm below (Algorithm 41).

**Algorithm 41** RefreshMasks

**Input:** $(x_i)$ for $0 \leq i \leq d$ with $\bigoplus_{i=0}^{d} x_i = x$

**Output:** $(x'_i)$ for $0 \leq i \leq d$ with $\bigoplus_{i=0}^{d} x'_i = x$

```
1: $x'_0 \leftarrow x_0$
2: for $i = 0$ to $d$ do
3:     $r_i \leftarrow \mathsf{rand}(2^n)$
4:     $x'_0 \leftarrow x'_0 \oplus r_i$
5:     $x'_i \leftarrow x_i \oplus r_i$
6: end for
7: return $(x'_i)_{0\leq i\leq d}$
```

### 6.1.2 Fast and Provably Secure Higher-order masking of AES

As already noted, the most costly operation in the implementation of the AES S-box is computing the multiplicative inverse over finite field $\mathbb{F}_{2^8}$. In order to accelerate the evaluation of inversion operation, several composite field methods were proposed [RDJ+01, SMTM01]. Kim *et al.,* [KHL11] used this idea to fasten the secure high-order masking of AES S-box proposed by Rivain-Prouff [RP10].

#### Composite Field.

A typical implementation of AES using composite field method follows the three step process [SMTM01]:

**Step 1.** Map all the elements from base filed to composite filed using an isomorphism function $\delta$.
**Step 2.** Compute the multiplicative inverse in the composite field.
**Step 3.** Map the result back to the base field from the composite filed using inverse isomorphism function $\delta^{-1}$.

every $2d$-tuple containing $d$ elements from the input $x$ $((x_i)_{1\leq i\leq d+1})$ and $d$ elements from the input $y$ $((y_i)_{1\leq i\leq d+1})$ should be uniformly distributed and independent of $x$ and $y$.

The composite field used in [SMTM01] is constructed using the following irreducible polynomials:

$$\begin{cases} \mathbb{F}_{(2^2)} & x^2+x+1 \\ \mathbb{F}_{(2^2)^2} & x^2+x+\phi \\ \mathbb{F}_{((2^2)^2)^2} & x^2+x+\gamma. \end{cases} \tag{6.2}$$

where $\phi = \{10\}_2$ and $\gamma = \{1100\}_2$.

For any element $A = a_h\gamma + a_l$ in composite field $\mathbb{F}_{((2^2)^2)^2}$, where $a_h, a_l \in \mathbb{F}_{(2^2)^2}$, the multiplicative inverse of $A$ can be computed as $A^{-1} = (A^{17})^{-1} \cdot A^{16}$. Here, $A^{16}$ can be computed by four bit wise XOR operations of $a_h\gamma + (a_h + a_l)$. The value $A^{17}$ can then be obtained by multiplying $A$ and $A^{16}$ over $\mathbb{F}_{((2^2)^2)^2}$, i.e., $A^{17} = \lambda a_h^2 + (a_h + a_l)a_l$ (since $\gamma^2 + \gamma = \lambda$). Hence, the inversion of $x \in \mathbb{F}_{2^8}$ can be computed by performing the following steps:

**Step 1.** Apply the isomorphism function $\delta : \mathbb{F}_{2^8} \to \mathbb{F}_{((2^2)^2)^2}$, such that $A = a_h\gamma + a_l = \delta(x)$, where $a_l, a_h \in \mathbb{F}_{((2^2)^2)^2}$.

$$\delta = \begin{pmatrix} 1&1&0&0&0&0&1&0 \\ 0&1&0&0&1&0&1&0 \\ 0&1&1&1&1&0&0&1 \\ 0&1&1&0&0&0&1&1 \\ 0&1&1&1&0&1&0&1 \\ 0&0&1&1&0&1&0&1 \\ 0&1&1&1&1&0&1&1 \\ 0&0&0&0&0&1&0&1 \end{pmatrix} \tag{6.3}$$

**Step 2.** Compute $A^{17}$ as $d = \lambda a_h^2 + (a_h + a_l)a_l \in \mathbb{F}_{2^4}$.
**Step 3.** Evaluate the inversion of $A^{17}$, namely, $d' = d^{-1}$.
**Step 4.** Compute the inversion $A^{-1} = (A^{17})^{-1} \cdot A^{16} = a'_h\lambda + a'_l$ where $a'_h = d'a_h \in \mathbb{F}_{2^4}$ and $a'_l = d(a_h + a_l) \in \mathbb{F}_{2^4}$.
**Step 5.** Compute the inversion of $x$ by applying the inverse mapping function $\delta^{-1} : \mathbb{F}_{((2^2)^2)^2} \to \mathbb{F}_{2^8}$, i.e., $x^{-1} = \delta^{-1}(a'_h\gamma + a'_l)$.

$$\delta^{-1} = \begin{pmatrix} 1&0&1&0&1&1&1&0 \\ 0&0&0&0&1&1&0&0 \\ 0&1&1&1&1&0&0&1 \\ 0&1&1&1&1&1&0&0 \\ 0&1&1&0&1&1&1&0 \\ 0&1&0&0&0&1&1&0 \\ 0&0&1&0&0&0&1&0 \\ 0&1&0&0&0&1&1&1 \end{pmatrix} \tag{6.4}$$

**Secure Inversion over Composite Field.**

Instead of securely raising an element to 254, the method from [KHL11] performs secure inversion by using composite field method, i.e., it securely masks the aforementioned five steps.

From (6.1) we already know that the linear functions $\delta$ and $\delta^{-1}$ can be masked by simply applying the function on each share separately. The field multiplication in $\mathbb{F}_{2^4}$ can be masked in the same way as shown in Algorithm 39. The multiplicative inversion in $\mathbb{F}_{2^4}$, i.e., raising the operand to 14, can be implemented as a combination of two linear operations (namely, squaring and raising to power 4) and one secure field multiplication, which is constructed as follows:

$$\begin{tikzcd} \boxed{x} \arrow[r, "x^2", "S"'] & \boxed{x^2} \arrow[r, "x^2x", "M"'] & \boxed{x^3} \arrow[r, "(x^3)^4", "2S"'] & \boxed{x^{12}} \arrow[r, "x^{12}x^2", "M"'] & \boxed{x^{14}} \end{tikzcd} \tag{6.5}$$

All these operations can be directly masked using the techniques proposed in [RP10]. We recall the algorithm to compute secure inversion in $\mathbb{F}_{2^4}$ in Algorithm 42.

**Algorithm 42** SecExp14

```
Input: $(x_i)$ for $0 \le i \le d$ with $\bigoplus_{i=0}^{d} x_i = x$
Output: $(y_i)$ for $0 \le i \le d$ with $\bigoplus_{i=0}^{d} y_i = x^{14}$
 1: for $i = 0$ to $d$ do
 2:     $z_i \leftarrow x_i^2$
 3: end for
 4: $(z_i)_{0\le i\le d} \leftarrow$ RefreshMasks $((z_i)_{0\le i\le d})$
 5: $(y_i)_{0\le i\le d} \leftarrow$ SecMult$((z_i)_{0\le i\le d}, (x_i)_{0\le i\le d})$
 6: for $i = 0$ to $d$ do
 7:     $w_i \leftarrow y_i^4$
 8: end for
 9: $(y_i)_{0\le i\le d} \leftarrow$ SecMult$((z_i)_{0\le i\le d}, (w_i)_{0\le i\le d})$
10: return $(y_i)_{0\le i\le d}$
```

We can see that we only need two calls to SecMult procedure here and hence is more efficient than Algorithm 40. To further improve the efficiency for implementing their solutions on embedded systems, the authors suggest to pre-compute several tables of 16 elements or 256 elements, such as field multiplication table, squaring table and isomorphism function table, which can significantly improve the overall performance. The running times can be further reduced by combining the inverse isomorphism function and affine function and storing the result in a table of size 256 bytes.

## 6.2 Higher-order Side-channel Security and Mask Refreshing

The security of Prouff-Rivain scheme and it's subsequent improvement of Kim *et al.* were proven using ISW model. The original result from ISW requires $2d + 1$ shares for an algorithm to be secure against attacks of order $d$. However, the result from Prouff and Rivain proved that it is sufficient to have $d+1$ shares provided the input shares are mutually independent of each other, thus improving the efficiency. For this purpose, they make use of RefreshMasks procedure which re-randomizes the input shares before every call to SecMult in case the inputs are not independent.

Though the procedures RefreshMasks and SecMult are secure against $d$-th order attacks independently, Coron *et al.* showed that their combination is insecure against attacks of order $\lfloor d/2 \rfloor + 1$. They also proposed a fix by adopting the ISW scheme to securely compute multiplications of the from $x \odot g(x)$, where $g$ is a linear function.

The attack from [CPRR13] combines $\lfloor d/2 \rfloor$ intermediate variables from the SecMult procedure and one intermediate variable from the RefreshMasks procedure. The authors also gave an attack simulation for second-order Mutual Information Analysis (MIA). Though the attacks were not highly successful in recovering the key (less than 20% success rate), a more powerful attacker could possibly recover larger portion of the secret key.

To avoid the attack the authors propose a solution avoiding the RefreshMasks procedure and proposed a method to securely compute multiplication on dependent inputs directly. Assume that the inputs to the SecMult are of the form $(x_i)_{1 \le i \le d+1}$ and $(y_i)_{1 \le i \le d+1} = g((x_i)_{1 \le i \le d+1})$ where $g$ is a linear function. This is possible since the inputs are dependent. Let function $f : \mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$ is defined as follows:

$$f(x, y) \quad = \quad (x \odot g(y)) \oplus (g(x) \odot y)$$

where $\odot$ denotes the filed multiplication. By the law of bilinearity, for every $x, y, r \in \mathbb{F}_{2^8}$ we have:

$$f(x, y) = f(x \oplus r, y) \oplus f(r, y) = f(x, y \oplus r) \oplus f(x, r)$$

In the SecMult procedure (Algorithm 39) the only place where the inputs are combined is in Step 4, which can rewritten with $f$ as follows:

$$r_{j,i} = (r_{i,j} \oplus (x_i \odot y_j)) \oplus (x_j \odot y_i) = f(x_i, x_j) \oplus r_{i,j} \tag{6.6}$$

Let $r'_{i,j}$ be a freshly generated random, then form bilinearity, we have

$$f(x_i, x_j) = f(x_i, x_j \oplus r'_{i,j}) \oplus f(x_i, r'_{i,j})$$

If we substitute this in 6.6, we get

$$r_{j,i} = (r_{i,j} \oplus f(x_i, r'_{i,j})) \oplus f(x_i, x_j \oplus r'_{i,j})$$

Hence by using freshly generated random and modifying Algorithm 39, we obtain a secure solution which computes $x \odot g(x)$, where $g$ is a linear function. We recall the corresponding algorithm below (Algorithm 43).

## 6.3 Improved Higher-order Secure Masking of AES S-Box

In this section we show that the attack from [CPRR13] (recalled in section 6.2) can be extended to the method proposed by Kim *et al.* as well [KHL11]. We also repair the original countermeasure which overcomes the attack.

**Algorithm 43** CombSecMult

```
Input: $(x_i)$ and $(y_i)$ for $0 \leq i \leq d$
Output: $(z_i)$ for $0 \leq i \leq d$, with $\bigoplus_{i=0}^{d} z_i = x \odot y$
 1: for $i = 0$ to $d$ do
 2:     for $j = i + 1$ to $d$ do
 3:         $r_{i,j} \leftarrow \mathsf{rand}(2^n)$                          ▷ Generate $n$-bit random
 4:         $r'_{i,j} \leftarrow \mathsf{rand}(2^n)$
 5:         $r_{j,i} \leftarrow (r_{i,j} \oplus (x_i \odot g(r'_{i,j})) \oplus (r'_{i,j} \odot g(x_i))$
 6:                 $\oplus(x_i \odot g(x_j \oplus r'_{i,j})) \oplus ((x_j \oplus r'_{i,j}) \odot g(x_i))$
 7:     end for
 8: end for
 9: for $i = 0$ to $d$ do
10:     $z_i = x_i \odot y_i$
11:     for $j = 1$ to $d$ do
12:         if $i \neq j$ then
13:             $z_i \leftarrow z_i \oplus r_{i,j}$
14:         end if
15:     end for
16: end for
17: return $(z_i)_{0 \leq i \leq d}$
```

The attack occurs when we combine the intermediate variables from the SecMult and RefreshMasks procedures *i.e.,* in step 4 and step 5 of Algorithm 42. For simplicity let us assume that $d$ is even. Let us consider the value of the intermediate variable $x'_0$ in the RefreshMasks procedure after $d/2$ iterations when called with $(z_i)_{0 \leq i \leq d}$:

$$\begin{aligned} z'_0 &= z_0 \oplus \bigoplus_{i=1}^{d/2} r_i \\ &= z \oplus \bigoplus_{i=1}^{d} z_i \oplus \bigoplus_{i=1}^{d/2} r_i \\ &= z \oplus \bigoplus_{i=1}^{d/2} (z_i \oplus r_i \oplus z_{d/2+i}) \\ &= g(x) \oplus \bigoplus_{i=1}^{d/2} (g(x_i) \oplus r_i \oplus g(x_{d/2+i})) \end{aligned}$$

Let us now consider the intermediate variables $(z'_i \odot x_j)_{0 \leq i \leq d/2; j=i+d/2}$ in step 4 of Algorithm 39 when called with $(z'_i)_{0 \leq i \leq d}$ and $(x_i)_{0 \leq i \leq d}$:

$$\begin{aligned}(y_i)_{0\le i\le d/2} &= z'_i \odot x_{i+d/2} \\ &= (z_i \oplus r_i) \odot x_{i+d/2} \\ &= (g(x_i) \oplus r_i) \odot x_{i+d/2}\end{aligned}$$

Now when we combine $(y_i)_{0\le i\le d/2}$ and $z'_0$ we can retrieve $z = g(x)$ as the variables $\bigoplus_{i=1}^{d/2}(g(x_i) \oplus r_i \oplus g(x_{d/2+i}))$ and $\bigoplus_{i=1}^{d/2}(g(x_i) \oplus r_i) \odot x_{i+d/2}$ are dependent on each other. Note that this attack is similar to the one proposed in [CPRR13].

**Our Improved Algorithm for Secure Inversion over Composite Field.**

In order to avoid the above attack, we propose a new secure inversion algorithm as shown in Algorithm 44 using SecMult1 procedure (Algorithm 43) over $\mathbb{F}_{2^4}$. It can be seen that the new algorithm is similar to 42 except that the calls to Refresh-Masks procedure and SecMult procedure are combined with a single call to SecMult1 procedure.

**Theorem 6.1.** *Algorithm 44 is secure against d-th order attacks.*

*Proof.* The security of Algorithm 44 directly follows from the proof given in Section 4 of [CPRR13] and hence is omitted. □

**Algorithm 44** SecInv4

```
Input: $(x_i)$ for $0 \le i \le d$ with $\bigoplus_{i=0}^{d} x_i = x$
Output: $(y_i)$ for $0 \le i \le d$ with $\bigoplus_{i=0}^{d} y_i = x^{14}$
1: for $i = 0$ to $d$ do
2:     $w_i \leftarrow x_i^2$                                                      ▷ $\bigoplus_i w_i = x^2$
3: end for
4: $(z_i)_{0\le i\le d} \leftarrow$ CombSecMult($(x_i)_{0\le i\le d}, (w_i)_{0\le i\le d}$)      ▷ $\bigoplus_i z_i = x^3$
5: for $i = 0$ to $d$ do
6:     $z_i \leftarrow z_i^4$                                                      ▷ $\bigoplus_i z_i = x^{12}$
7: end for
8: $(y_i)_{0\le i\le d} \leftarrow$ CombSecMult($(z_i)_{0\le i\le d}, (w_i)_{0\le i\le d}$)      ▷ $\bigoplus_i y_i = x^{14}$
9: return $(y_i)_{0\le i\le d}$
```

**Implementation Results**

We implemented our improved scheme as well as Coron *et al.* [CPRR13] scheme on an ARM NEON processor. NEON is an advanced SIMD (Single Instruction Multiple Data) extension to some of the ARM processors. With the help of rich instruction set provided by NEON and data parallelism, we could achieve significant speed up for higher-order masking.

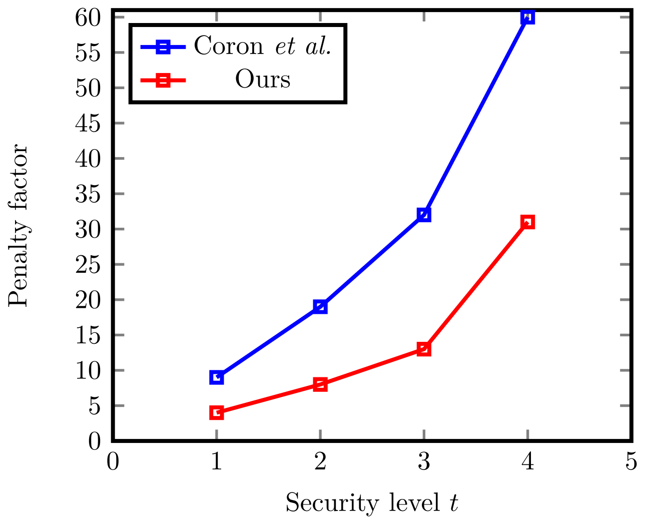


Figure 6.1: Penalty factor for different HO-secure masked implementations of AES

We compared our implementation results with existing countermeasures from CHES 2010 [RP10], CHES 2011 [KHL11] and Eurocrypt 2014 [Cor14]. For all the schemes, we considered the security against attacks of order upto 4. We compared the penalty factor for each of these schemes with respect to the unmasked AES implementation. We report the corresponding results in Table 6.1. With our proposed technique, the second and third order secure AES is only 8 and 13 times slower than the unmasked implementation as shown in Figure 6.1. Moreover, our results achieve a speedup factor of three compared with the fastest solutions available.

Table 6.1: Penalty factor for different HO masked implementations of AES

| Method | first-order | second-order | third-order | fourth-order |
|---|---|---|---|---|
| CHES'10 [RP10] | 65 | 132 | 235 | - |
| CHES'11 [KHL11] | - | 22 | 39 | - |
| Coron [Cor14] | 439 | 1205 | 2411 | 4003 |
| Coron *et al.* [CPRR13] (NEON) | 9 | 19 | 32 | 60 |
| Our scheme (NEON) | 4 | 8 | 13 | 31 |

# Chapter 7

# Conversion of Security Proofs from One Model to Another: A New Issue

This chapter presents a new issue concerning the security proofs of masking schemes. In general security of a masking scheme is proven in Only manipulated Data Leakage (ODL) model which assumes that the leakage from a device depends only on the manipulated data at a particular instance in time (*e.g.* Hamming weight). However, in practice the leakage in some implementations is better modeled using memory transition leakage (MTL) model, where the leakage is dependent on the transitions on the memory bus (*e.g.* Hamming distance). This raises the question: how secure are the masking schemes proven in ODL model when the model is changed to MTL? We answer this question in the negative. We show that a second-order masking scheme secure in ODL model can be broken using a first-order attack in MTL model. We also show that the straightforward approach of erasing the memory contents before every write operation does not work as well. This result emphasizes the need to re-evaluate the security while porting an implementation from ODL model to MTL model. This is a joint work with Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner and Matthieu Rivain. A part of this work appeared in the proceedings of COSADE, 2012 [CGP+12b].

**Contents**

## 7.1 Introduction

Two different models are considered prominently in the literature to prove masking. We recall these models below.

1. **ODL Model:** This leakage model assumes that the device leakage is a function of the manipulated data. Assume that an intermediate variable $Z$ is being manipulated by the device, then the leakage $L$ satisfies:

$$L = \varphi(Z) + B \tag{7.1}$$

   where $\varphi$ is a (non-constant) function and $B$ an independent Gaussian noise with zero mean. Examples of such leakage functions include the Hamming weight (HW) function (or an affine function of the HW), in which case the model is also referred to as *Hamming weight model.* Another example can be the identity function *i.e.* the leakage reveals the value of $Z$.

2. **MTL Model:** This leakage model assumes that the device leaks on the *memory transitions* when a value $Z$ is manipulated. Here, the function $\varphi$ depends on the current value $Z$ as well as the initial value in the memory $Y$. Namely, we have:

$$L = \varphi(Z \oplus Y) + B. \tag{7.2}$$

   In the particular case where $\varphi$ is the HW function, the leakage $L$ defined in (7.2) corresponds to the so-called Hamming distance (HD) model.

Several works have demonstrated the validity of HW and HD models in practice, which are today commonly accepted by the SCA community. However other more precise models exist in the literature (see for instance [SLP05, PSQ07, DPRS11]). In the rest of this chapter, we keep the generality by considering two models : ODL model (*Only manipulated Data Leak*) and MTL model (*Memory Transition Leak*), each of them being defined by the leakage function expressed in (7.1) and (7.2) respectively.

### 7.1.1 ODL Model *vs.* MTL Model

In general security proofs are given in ODL model due to it's simplicity. The same is the case with second-order secure masking scheme proposed by Prouff and Rivain in [RDP08]. However, leakage of several devices can be better modeled using MTL model (especially in ASIC implementations). Starting from this observation, a natural question is to decide at which extent a countermeasure proved to be secure in

ODL model stays secure in MTL model. Similarly, another interesting and practically relevant problem is the design of methods to transform an implementation secure in the first model into a new implementation secure in the second. Hence, if we assume that the memory transitions leak information, the leakage is modeled by $\varphi(Y \oplus Z) + B$. In such a model a masking countermeasure may become ineffective. For instance, if $Z$ corresponds to a masked variable $X \oplus M$ and if $Y$ equals the mask, then the leakage reveals information on $X$. A very straightforward idea to deal with this issue is to erase the memory before each new writing (*e.g.* set $Y$ to 0 in our example). One may note that such a technique is often used in practice at either the hardware or software level. Using such a method, the leakage $\varphi(Y \oplus Z) + B$ is replaced by the sequence of consecutive leakages $\varphi(Y \oplus 0) + B_1$ and $\varphi(0 \oplus Z) + B_2$ that is equivalent to $\varphi(Y) + B_1$ and $\varphi(Z) + B_2$. The single difference with classical ODL model is the additional assumption that the execution leaks the content of the memory before the writings. Since this leakage corresponds to a variable that has been manipulated prior to $Z$, it is reasonable to assume that the leakage $\varphi(Y) + B_1$ has already been taken into account when establishing the security of the countermeasure. As a consequence, this way to implement a countermeasure proved to be secure in ODL model seems at a first glance also offers security on a device leaking in MTL model.

In this chapter, we emphasize that a countermeasure proved to be secure in ODL model may no longer stay secure in MTL model. Indeed, we exhibit a case where a countermeasure proved to be second-order resistant in ODL model does no longer provide security against first-order SCA when implemented in a device leaking on the memory transitions. Then, we show that the natural method proposed above to transfer a countermeasure resistant in ODL model into a countermeasure resistant in MTL model is flawed. Those two results enlighten the actual lack of a framework to solve the (practically) important issue of porting an implementation from one family of devices to the other one.

### 7.1.2 Organization

This chapter is organized as follows. In Section 7.2, we briefly recall a second-order countermeasure proved to be secure in ODL model [RDP08]. In Section 7.3, we show that such a countermeasure can be broken by using a first-order attack in MTL model. To thwart this attack, we apply in Section 7.4 the method described previously which erases the memory before each new writing and we show that this method is still insecure against second-order attacks. We provide the results of a practical implementation of our attacks in Section 7.5.

## 7.2 Securing Block Cipher Against 2O-SCA

Application of masking is straightforward for linear functions, which can be computed separately for each of the shares. For non-linear functions such as s-boxes, we can use two methods. In the first method one evaluates the s-box function on the fly by masking all the operations independently. This works based on the algebraic structure of the s-box and hence must be devised independently for each s-box. The second method consists in pre-computing a randomized s-box lookup table in RAM for every new execution of the algorithm and then mask the table. This technique is generic and hence can be applied to any s-box.

At FSE 2008 Prouff and Rivain proposed two generic methods to secure s-box lookup table against attacks of first and second-order [RDP08]. Their first method requires an intermediate table of same size (stored in the RAM), where as the second method avoids this table with an increase in the execution time. These algorithms were recalled in Section 2.4.1. Their second algorithm is given below (Algorithm 45) for quick reference.

**Algorithm 45** Prouff-Rivain Sec20-masking

**Input:** A masked value $\tilde{x} = x \oplus t_1 \oplus t_2 \in \mathbb{F}_{2^n}$, the pair of input masks $(t_1, t_2) \in \mathbb{F}_{2^n} \times \mathbb{F}_{2^n}$, a pair of output masks $(s_1, s_2) \in \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$, a $(n, m)$ s-box function $F$

**Output:** The masked S-box output $F(x) \oplus s_1 \oplus s_2 \in \mathbb{F}_{2^m}$

```
1: $b \leftarrow \mathsf{rand}(1)$
2: for $a = 0$ to $2^n - 1$ do
3:     $cmp \leftarrow compare_b(t_1 \oplus a, t_2)$
4:     $R_{cmp} \leftarrow (F(\tilde{x} \oplus a) \oplus s_1) \oplus s_2$
5: end for
6: return $R_b$
```

To compute $F(x) \oplus s_1 \oplus s_2$, the core idea of Algorithm 45 is to successively read all values of the lookup table $F$ from index $\tilde{x} \oplus a$ with $a = 0$ to index $\tilde{x} \oplus a$ with $a = 2^n - 1$. When the correct value $F(x) \oplus s_1 \oplus s_2$ is accessed, it is stored in a pre-determined register $R_b$ whereas the other values $F(\tilde{x} \oplus a) \oplus s_1 \oplus s_2$, with $\tilde{x} \oplus a \neq x$, are stored in a garbage register $R_{\overline{b}}$ . In practice two registers $R_0$ and $R_1$ are used and their roles are chosen thanks to a random bit $b$.

Depending on the loop index $a$, the fourth step of Algorithm 45 processes the following operation (as given in Algorithm 3):

$$\begin{cases} cmp \leftarrow b \ ; & R_{cmp} \leftarrow F(x) \oplus s_1 \oplus s_2 & \text{if } a = t_1 \oplus t_2 \\ cmp \leftarrow \overline{b} \ ; & R_{cmp} \leftarrow F(\tilde{x} \oplus a) \oplus s_1 \oplus s_2 & \text{otherwise} \end{cases} . \tag{7.3}$$

In view of (7.3), it may be observed that the register $R_b$ is modified only once whereas $R_{\overline{b}}$ changes $2^n - 1$ times. As proven in [RDP08], this behavior difference between the registers $R_b$ and $R_{\overline{b}}$ cannot be successfully exploited by a second-order attack when the device leaks in the ODL model. The proof can be straightforwardly extended to any leakage model called *linear*, in which all bits of the manipulated data leak independently. However, if Algorithm 45 must be implemented on a physical device with a different leakage model, then the security proof in [RDP08] can no longer be invoked. Hence, since the most common alternative is MTL model, it is particularly interesting to investigate whether Algorithm 45 stays secure in this context. In the next section, we put forward the kind of security issues brought by a straightforward implementation of Algorithm 45 on a device leaking the memory transition. In particular, for a specific (but quite natural) implementation, we exhibit a first-order SCA.

## 7.3 Attack of Algorithm 45 in the MTL Model

This section is organized as follows: first we present a straightforward implementation of the 2O-SCA countermeasure described in Algorithm 45. Then we expose how a first-order attack in MTL model can break this second-order countermeasure.

In the analysis developed in this chapter, we will denote random variables by capital letters (*e.g.* $X$) and their values by small letters (*e.g.* $x$).

### 7.3.1 Straightforward Implementation of Algorithm 45

In the following, we assume that the considered device is based on an assembler language for which a register $R_A$ is used as accumulator. Moreover we assume that registers $R_A$, $R_0$ and $R_1$ are initially set to zero.

Based on these assumptions, the fourth step of Algorithm 45 can be implemented in the following way:

$$\begin{array}{lll} 4.1 & R_A & \leftarrow \tilde{x} \oplus a \\ 4.2 & R_A & \leftarrow F(R_A) \\ 4.3 & R_A & \leftarrow R_A \oplus s_1 \\ 4.4 & R_A & \leftarrow R_A \oplus s_2 \\ 4.5 & R_{cmp} & \leftarrow R_A \end{array} \tag{7.4}$$

During this processing where $\tilde{X} = X \oplus T_1 \oplus T_2$, the initial content of register $R_{cmp}$, denoted by $Y$, satisfies the following equation depending on the values of the loop index $a$, $T_1$ and $T_2$:

$$Y = \begin{cases} 0 & \text{if } a = 0 \ , \\ 0 & \text{if } a = 1 \text{ and } T_1 \oplus T_2 = 0 \ , \\ 0 & \text{if } a > 0 \text{ and } T_1 \oplus T_2 = a \ , \\ F(\tilde{X} \oplus (a-2)) \oplus S_1 \oplus S_2 & \text{if } a > 1 \text{ and } T_1 \oplus T_2 = (a-1) \ , \\ F(\tilde{X} \oplus (a-1)) \oplus S_1 \oplus S_2 & \text{otherwise.} \end{cases} \tag{7.5}$$

In the following we will show that the distribution of the value $Y$ defined in (7.5) brings information on the sensitive variable $X$. We will consider two cases depending on whether $R_A$ equals $R_{cmp}$ or not.

### 7.3.2 Description of the First-order Attack when $R_A = R_{cmp}$

According to this decomposition, if we assume that the register $R_{cmp}$ is the accumulator register, then Step 4.5 of (7.4) is unnecessary and the register $R_{cmp}$ leaks at each state. This is in particular the case at Step 4.1,

In this part, we assume that the physical leakage of the device is modeled by MTL model and hence the leakage $L$ associated to Step 4.1 of (7.4) satisfies:

$$L \sim \varphi(Y \oplus \tilde{X} \oplus a) + B \ , \tag{7.6}$$

where $Y$ denotes the initial state of $R_{cmp}$ before being updated with $\tilde{X} \oplus a$, defined above by (7.5).

From (7.5) and (7.6), we deduce:

$$L = \begin{cases} \varphi(\tilde{X}) \quad + B & \text{if } a = 0 \ , \\ \varphi(X \oplus 1) + B & \text{if } a = 1 \ \& \ T_1 \oplus T_2 = 0 \ , \\ \varphi(X) \quad + B & \text{if } a > 0 \ \& \ T_1 \oplus T_2 = a \ , \\ \varphi(F(\tilde{X} \oplus (a-2)) \oplus S_1 \oplus S_2 \oplus \tilde{X} \oplus a) + B & \text{if } a > 1 \ \& \ T_1 \oplus T_2 = (a-1) \ , \\ \varphi(F(\tilde{X} \oplus (a-1)) \oplus S_1 \oplus S_2 \oplus \tilde{X} \oplus a) + B & \text{otherwise.} \end{cases} ,$$

When $a = 0$, the leakage $L$ is an uniform value which brings no information on the value $X$. Therefore in the following, we omit this particular case.

Hence, we have

$$L = \begin{cases} \varphi(X) & +B & \text{if } T_1 \oplus T_2 = a \ , \\ \varphi(X \oplus 1) & +B & \text{if } T_1 \oplus T_2 = 0 \ \text{ and } \ a = 1 \ , \\ \varphi(Z) & +B & \text{otherwise} \ , \end{cases} \tag{7.7}$$

with $Z$ a variable independent of $X$ and with uniform distribution.

In view of (7.7), the leakage $L$ depends on $X$. Indeed, the mean of $(L|X = x)$ satisfies:

$$E(L \mid X = x) = \begin{cases} \frac{1}{2^n} \times (\varphi(x) + \varphi(x \oplus 1)) + \frac{2^n-2}{2^n} \times E(\varphi(Z)) & \text{if } a = 1 \ , \\ \\ \frac{1}{2^n} \times \varphi(x) + \frac{2^n-1}{2^n} \times E(\varphi(Z)) & \text{if } a > 1 \ , \end{cases}$$

or equivalently (since $Z$ has uniform distribution):

$$E(L \mid X = x) = \begin{cases} \frac{1}{2^n} \times (\varphi(x) + \varphi(x \oplus 1)) + \frac{n \times (2^n-2)}{2^{n+1}} & \text{if } a = 1 \ , \\ \\ \frac{1}{2^n} \times \varphi(x) + \frac{n \times (2^n-1)}{2^{n+1}} & \text{if } a > 1 \ . \end{cases} \tag{7.8}$$

When $a > 1$, the mean in (7.8) is an affine function of $\varphi(x)$ and it is an affine function of $(\varphi(x) + \varphi(x \oplus 1))$ otherwise. Therefore in both cases the mean leakage

reveals some information on $X$.
An adversary can thus target the second round in Algorithm 45 (*i.e.* $a = 1$) and get a sample of observations for the leakage $L$ defined as in (7.6). The value $X$ typically corresponds to the bitwise addition between a secret sub-key $K$ and a known plaintext subpart $M$. In such a case and according to the statistical dependence shown in (7.8), the set of observations can be used to perform a first-order SCA allowing an attacker to recover the secret value $K$.

As an illustration, we simulated a first-order CPA in the Hamming weight model without noise targeting the second loop (namely $a = 1$) with the AES S-box. The secret key byte was recovered with a success rate of 99% by using 1.000.000 acquisitions.

### 7.3.3 Description of the First-order Attack when $R_A \neq R_{cmp}$

In this part, the accumulator register $R_A$ is assumed to be different from the register $R_{cmp}$. Under such an assumption, Step 4.5 in (7.4) leaks the transition between the initial content $Y$ of $R_{cmp}$ and the current content of $R_A$. Namely, after denoting $T_1 \oplus T_2$ and $S_1 \oplus S_2$ by $T$ and $S$ respectively, we have:

$$L = \varphi(Y \oplus F(X \oplus T \oplus a) \oplus S) + B. \tag{7.9}$$

Due to (7.5), Relation (7.9) may be developed in the following way according to the values of $a$ and $T$:

$$L = \begin{cases} \varphi(\ F(X \oplus T) \oplus S\ ) + B & \text{if } a = 0, \\ \varphi(\ F(X) \oplus S\ ) + B & \text{if } a = 1 \text{ and } T = (a-1), \\ \varphi(\ F(X) \oplus S\ ) + B & \text{if } a > 0 \text{ and } T = a, \\ \varphi(\ D_{a \oplus (a-2)} F(X \oplus (a-2) \oplus (a-1)\ ) + B & \text{if } a > 1 \text{ and } T = (a-1), \\ \varphi(\ D_{a \oplus (a-1)} F(X \oplus (a-1) \oplus T)\ ) + B & \text{otherwise,} \end{cases} \tag{7.10}$$

where $D_y F$ denotes the *derivative* of $F$ *with respect to* $y \in \mathbb{F}_2^n$, which is defined for every $x \in \mathbb{F}_2^n$ by $D_y F(x) = F(x) \oplus F(x \oplus y)$.

In the three first cases in (7.10), the presence of $S$ implies that the leakage $L$ is independent of $X$. Indeed, in these cases the leakage is of the form $\varphi(Z) + B$ where $Z$ is an uniform random variable independent of $X$. In the last two cases, $S$ does not appear anymore. As a consequence it may be checked that the leakage $L$ depends on $X$. Indeed, due to the law of total probability, for any $x$ and $a = 1$, the mean of $(L|X = x)$ satisfies:

$$E(L|X = x) = \frac{2\mu}{2^n} + \frac{1}{2^n} \sum_{t=2}^{2^n - 1} \varphi(D_a F(x \oplus t)), \tag{7.11}$$

where $\mu$ denotes the expectation $E[\varphi(U)]$ with $U$ uniform over $\mathbb{F}_{2^n}$ (*e.g.* for $\varphi = \mathrm{HW}$

we have $\mu = n/2$). And when $a > 1$, the mean of $(L|X = x)$ satisfies:

$$E(L|X=x) = \frac{\mu}{2^n} + \frac{1}{2^n}\,\varphi(D_{a\oplus(a-2)}F(x \oplus (a-2) \oplus (a-1)))$$

$$+\frac{1}{2^n}\sum_{t=0,t\neq a,(a-1)}^{2^n-1} \varphi(D_{a\oplus(a-1)}F(x \oplus (a-1) \oplus t)). \qquad (7.12)$$

From an algebraic point of view, the sums in (7.11) and (7.12) may be viewed as the mean of the value taken by $D_aF(x\oplus t)$ (respectively $D_{a\oplus(a-1)}F(x\oplus(a-1)\oplus t)$) over the coset $x\oplus\{t, t\in[2, 2^n-1]\}$ (respectively $x\oplus\{t, t\in[0, 2^n-1]\setminus\{a-1, a\}\}$). Since those cosets are not all equal, the means are likely to be different for some values of $x$. Let us for instance consider the case of $F$ equal to the AES S-box and let us assume that $\varphi$ is the identity function. In Relation (7.11), the sum equals 34066 if $x = 1$ and equals 34046 if $x = 2$. When $a > 1$, we have the similar observation.

From (7.11) and (7.12), we can deduce that the mean leakage reveals information on $X$ and thus, the set of observations can be used to perform a first-order SCA.

By exhibiting several attacks in this section, we have shown that the second-order countermeasure proved to be secure in ODL model may be broken by a first-order attack in MTL model. These attacks demonstrate that a particular attention must be paid when implementing Algorithm 45 on a device leaking in MTL model. Otherwise, first-order leakage may occur as those exploited in the attacks presented above. As already mentioned in the introduction, a natural solution to help the security designer to deal with those security traps could be to systematically erase the registers before any writing. This solution is presented and discussed in the next section.

## 7.4 Study of a Straightforward Patch

In the following, we present a straightforward method to patch the flaw exhibited in the previous section. The aim of this patch is to transform an implementation secure in ODL model into an implementation secure in MTL model. It essentially consists in erasing the memory before each new writing. In this section, we evaluate this strategy when applied to implement Algorithm 45 leaking in MTL model. Then, we show that this natural method does not suffice to go from security in ODL model to security in MTL model. Indeed, we present a second-order attack against the obtained second-order countermeasure.

### 7.4.1 Transformation of Algorithm 45 into a MTL-Resistant Scheme

As in the previous section, we assume that the leakage model is MTL model and that the registers $R_b$ and $R_{\overline{b}}$ are initially set to zero. In order to preserve the security proof given in the first model, we apply a solution consisting in erasing the memory before each new writing.

Based on these assumptions, the fourth step of Algorithm 45 can be implemented in the following way:

$$\begin{array}{lll} 4.1 & R_{cmp} & \leftarrow 0 \\ 4.2 & R_{cmp} & \leftarrow F(\tilde{x} \oplus a) \oplus s_1 \oplus s_2 \end{array} \tag{7.13}$$

As previously, we assume that the initial state of $R_{cmp}$ before Step 4.1 is equal to $Y$. Then, according to this decomposition, the register $R_{cmp}$ is set to 0 before the writing of $Z = F(\tilde{X} \oplus a) \oplus S_1 \oplus S_2$ in the Step 4.2. Hence, the leakage defined by (7.6) is replaced by the sequence of consecutive leakages $\varphi(Y, 0) + B_1$ (Step 4.1), $\varphi(0, Z) + B_2$ (Step 4.2), that is $\varphi(Y) + B_1$, $\varphi(Z) + B_2$. However this model is not equivalent to the ODL model since here the previous value in $R_{cmp}$ leaks whenever it is erased. And as we show hereafter, such a leakage enables a second-order attack breaking the countermeasure although secure in the ODL model.

### 7.4.2 Description of a Second-order Attack

To perform our second-order attack, we use two information leakages $L_1$ and $L_2$ during the same execution of Algorithm 45 implemented with (7.13).

The first leakage $L_1$ corresponds to the manipulation of $\tilde{X}$ prior to Algorithm 45. $L_1$ thus satisfies:

$$L_1 \sim \varphi(\tilde{X}) + B_0. \tag{7.14}$$

The second leakage $L_2$ corresponds to Step 4.1 of (7.13). Thus it satisfies:

$$L_2 \sim \varphi(Y) + B_1. \tag{7.15}$$

From (7.5) and (7.15), we deduce:

$$L_2 = \begin{cases} \varphi(0) + B_1 & \text{if } a = 0 \ , \\ \varphi(0) + B_1 & \text{if } a = 1 \text{ and } T_1 \oplus T_2 = 0 \ , \\ \varphi(0) + B_1 & \text{if } a > 0 \text{ and } T_1 \oplus T_2 = a \ , \\ \varphi(F(\tilde{X} \oplus (a-2)) \oplus S_1 \oplus S_2) + B_1 & \text{if } a > 1 \text{ and } T_1 \oplus T_2 = (a-1) \ , \\ \varphi(F(\tilde{X} \oplus (a-1)) \oplus S_1 \oplus S_2) + B_1 & \text{otherwise.} \end{cases} \tag{7.16}$$

which implies that:

$$L_2 = \begin{cases} \varphi(0) + B_1 & \text{if } a = 0 \ , \\ \varphi(0) + B_1 & \text{if } a = 1 \text{ and } T_1 \oplus T_2 = 0 \text{ or } 1 \ , \\ \varphi(Z) + B_1 & \text{if } a = 1 \text{ and } T_1 \oplus T_2 \neq 0 \text{ or } 1 \ , \\ \varphi(0) + B_1 & \text{if } a > 1 \text{ and } T_1 \oplus T_2 = a \ , \\ \varphi(Z) + B_1 & \text{if } a > 1 \text{ and } T_1 \oplus T_2 \neq a \ , \end{cases} \tag{7.17}$$

where $Z$ is a variable independent of $X$ and with uniform distribution.

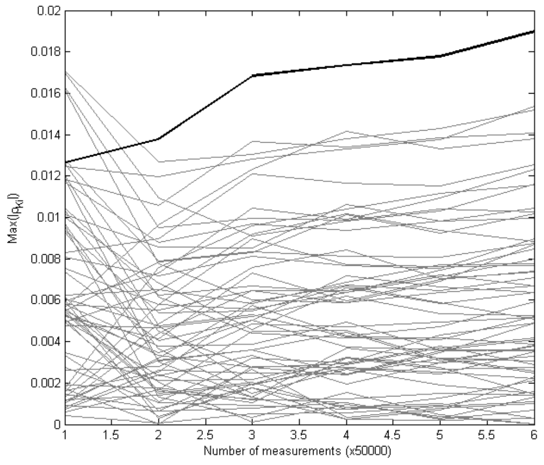


Figure 7.1: Convergence with simulated curves without noise, for $a = 1$.

From (7.17), the leakage is independent from $T_1 \oplus T_2$ when $a = 0$. For this reason, in the following we only study the mean of $L_2$ for $a > 0$:

$$E(L_2) = \begin{cases} \varphi(0) & \text{if } a = 1 \text{ and } T_1 \oplus T_2 = 0 \text{ or } 1 \ , \\ \varphi(Z) & \text{if } a = 1 \text{ and } T_1 \oplus T_2 \neq 0 \text{ or } 1 \ , \\ \varphi(0) & \text{if } a > 1 \text{ and } T_1 \oplus T_2 = a \ , \\ \varphi(Z) & \text{if } a > 1 \text{ and } T_1 \oplus T_2 \neq a \ , \end{cases}$$

or equivalently (since $Z$ has uniform distribution):

$$E(L_2) = \begin{cases} \varphi(0) & \text{if } a = 1 \text{ and } T_1 \oplus T_2 = 0 \text{ or } 1 \ , \\ \varphi(0) & \text{if } a > 1 \text{ and } T_1 \oplus T_2 = a \ , \\ \frac{n}{2} & \text{otherwise.} \end{cases} \quad (7.18)$$

On the other hand, the leakage $L_1$ depends by definition on $X \oplus T_1 \oplus T_2$. As a consequence, one deduces that the pair $(L_1, L_2)$ statistically depends on the sensitive value $X$. Moreover, it can be seen in (7.18) that the leakage on $T_1 \oplus T_2$ is maximal when $a = 1$. An adversary can thus target the second loop in Algorithm 45 (*i.e.* $a = 1$), make measurements for the pair of leakages $(L_1, L_2)$ and then perform a 2O-CPA to extract information on $X$ from those measurements.

We have simulated such a 2O-SCA with $X = M \oplus K$ where $M$ is a 8-bit value known to the attacker and $K$ a 8-bit secret key value. By combining $L_1$ and $L_2$ using the normalized multiplication and the optimal prediction function as defined in [PRB09a], the secret value $k$ is recovered with a success rate of 99% by using less than 200.000 curves. Fig.1 represents the convergence of the maximal correlation value for different key guesses over the number of leakage measurements. Each curve corresponds to some hypothesis on the secret $K$. In particular the black curve corresponds to the correct hypothesis $k$.

The second-order attack presented in this section show that erasing registers before writing a new value does not suffice to port the security of an implementation from ODL model to MTL model. For the case of Algorithm 45, a possible patch is to erase $R_{cmp}$ using a random value. However, though this patch works in the particular case of Algorithm 45, it does not provide a generic method to transform a $d$th-order countermeasure secure in the ODL model to a $d$th-order countermeasure secure in the MTL model. The design of such a generic method is an interesting problem that we leave open for future research.

## 7.5 Experimental Results

This section provides the practical evaluation of the attacks presented above. We have verified the attacks on block ciphers with two different kinds of S-boxes: an 8-bit to 8-bit S-box (AES) and two 4-bit to 4-bit S-boxes (PRESENT and Klein). We have implemented Algorithm 45 as described in Section 7.2 on a 8-bit micro-controller. Using 2O-CPA, we were able to find the secret key for all three S-boxes. In case of the $4 \times 4$ S-boxes, we needed fewer than 10.000 power traces to find the correct key. However, for the $8 \times 8$ S-box, the number was much higher, since more than 150.000 traces were required to distinguish the correct key from the rest of the key guesses.

Initially, we set the value in the two memory locations $R_0$ and $R_1$ to zero. We randomly generate the plaintexts $m_i$ and the input/output masks $t_{i,1}, t_{i,2}$ and $s_{i,1}, s_{i,2}$ using a uniform pseudo-random number generator where the value of $i$ varies from 1 to $N$ (i.e., the number of measurements). Then, we calculate $\tilde{x_i}$ from the correct key $k$ via $\tilde{x_i} = k \oplus m_i \oplus t_{i,1} \oplus t_{i,2}$. As described in Section 7.4, before writing a new value to any memory location, we first erase its contents by writing 0, and then write the new value as shown in (7.13). For verifying the attacks, we only consider the power traces where $a = 1$. During respectively the manipulation of the $\tilde{x}_i$ and the memory erasing, we measure the power consumption of the device. This results in a sample of pairs of leakage points that are combined thanks to the centered product combining function defined in [PRB09a]. For each key hypothesis $k_j$, the obtained combined leakage sample $(\mathcal{L}_j)_i$ is correlated with the sample of hypotheses $(HW(m_i \oplus k_j))_i$. The key guess for which the correlation coefficient is the maximum will be the correct key.

Figure 7.2 and Figure 7.3 show the correlation traces for a 2O-CPA on the Klein and PRESENT S-boxes, respectively. As it can be observed, the right key is found in both cases with less than 10.000 power traces. Figure 7.4 shows the correlation traces for a 2O-CPA on the AES S-box. Here the convergence of the traces to the correct key is observable only after 150.000 traces. Finally, Figure 7.5 shows the first-order attack on the PRESENT S-box in the Hamming Distance model as described in Section 7.3. Here we implemented Algorithm 1 directly without the additional step of erasing the memory contents before performing a write operation. The power traces are collected for 50.000 inputs, and only the traces corresponding

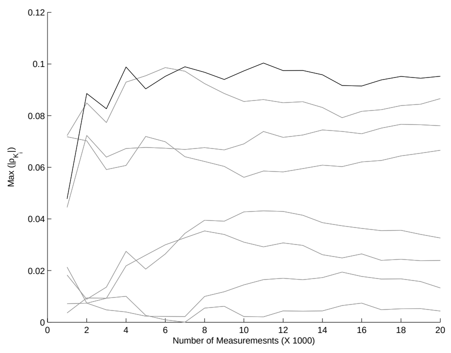


Figure 7.2: Convergence with practical implementation of 20-CPA for Klein.

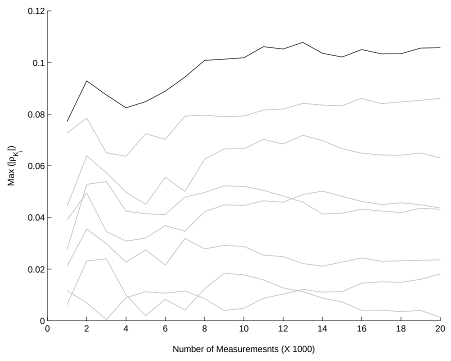


Figure 7.3: Convergence with practical implementation of 20-CPA for PRESENT

to the case $a = 1$ are considered. The correct key candidate can be identified with less than 10.000 traces.

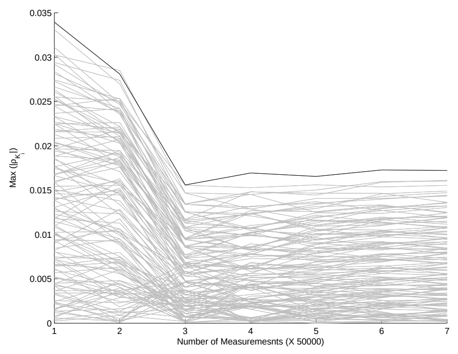


Figure 7.4: Convergence with practical implementation of 20-CPA for AES.

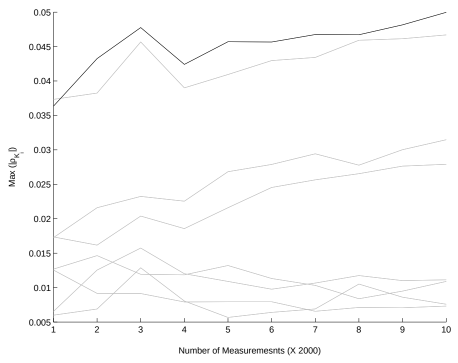


Figure 7.5: Convergence with practical implementation of 10-CPA for PRESENT.

# Chapter 8

# Conclusions

In chapter 3, we addressed the problem of secure conversion between Boolean and arithmetic masking for any order. By applying the ISW framework and Goubin's results for first-order conversion, we developed two algorithms of the same asymptotic complexity to securely add Boolean shares. We then described novel conversion algorithms between Boolean and arithmetic masking that are provably secure at any order. Practical experiments based on HMAC-SHA-1 as case study show that, in the case of second and third-order security, using Boolean masking and performing secure addition on Boolean shares directly is more efficient than converting between Boolean and arithmetic masking. Even though the proposed algorithms entail a massive performance penalty, they can still be practically useful for applications like challenge-response authentication where only a single block of data needs to be encrypted.

In chapter 4, we have described a new conversion algorithm from arithmetic to Boolean masking with complexity $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$ for Goubin's algorithm. We have also described a variant for performing the arithmetic addition modulo $2^k$ directly with Boolean shares, still with complexity $\mathcal{O}(\log k)$ instead of $\mathcal{O}(k)$. We then used the similar techniques to improve the complexity of higher-order conversion algorithms from $\mathcal{O}(n^2k)$ to $\mathcal{O}(n^2 \log k)$ for security against attacks of order $n$. In practice, for arithmetic additions modulo $2^{32}$ as in case of HMAC-SHA-1, we obtain similar performances as Goubin's algorithms and Debraize's algorithm and better performances for additions modulo $2^{64}$.

In chapter 5, we proposed time-memory trade-off solutions for conversion between Boolean and arithmetic masking for first and second-order. For second- order conversion, we improved the number of shares required from 5 to 3 when compared to CGV method. Our first conversion algorithms were obtained by adapting the second-order generic countermeasure due to Rivain *et al.* However these algorithms become infeasible for conversion size of greater than 10 bits. We then improved our solution using divide and conquer. We have shown that our improved algorithms reduce the required time by 85% with negligible memory overhead (around 16 bytes).

One open issue here is to find a way to perform addition on Boolean shares directly, which is secure against attacks of second-order. We can not apply the generic method of [RDP08] in this case because the S-box is not balanced. Such an S-box would require input of size $2l + 1$-bit ($l$-bit for each of the two arguments to addition and one bit for input carry) and output the $l + 1$-bit sum including the carry. For this function to be balanced, each of the $2^{l+1}$ possible outputs must be an image of exactly $2^l$ elements. However, this is not true since the element 0 can be image of only one element i.e., 0 and hence we can mount a second-order attack. Finding a solution to this problem could further improve the performance of second-order masking.

In chapter 6 we examined the fast and provably secure higher-masking of AES S-box proposed by Kim *et al.* at CHES 2011 [KHL11]. We have shown that their $n$-th order secure scheme is actually insecure against attacks of order $n/2 + 1$. We then gave an improved method which also fixes the flaw.

In chapter 7, we have shown that a particular attention must be paid when implementing a countermeasure proved to be secure in one model on devices which leak in another model. In particular we have shown that a second-order countermeasure with security proof in ODL model is broken by using first-order SCA when running on a device leaking in MTL model. Then, we have focused on a method that looked at first glance very natural to a scheme resistant in ODL model and shown that it doesn't work as well. Our analysis pointed out flaws in the conversion method and hence led us to identify two new issues that we think to be very promising for further research. Firstly, it is interesting to obtain a generic countermeasure proved to be secure in any practical model. Secondly, the design of a method of porting the security from a model to another one is also worth exploring.

# Bibliography

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side-Channel(s). In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop Proceedings*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2003. 5, 6

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, 3rd International Workshop Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001. 26

[Aum07] Sébastien Aumônier. Generalized Correlation Power Analysis . In *e-Proceedings of the Ecrypt Workshop Tools For Cryptanalysis*, 2007. 24

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004, 6th International Workshop Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004. 8

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997. 5

[BGG$^+$14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the Cost of Lazy Engineering for Masked Software Implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014. 19

[BGK05] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably Secure Masking of AES. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography - SAC,*, volume 3357 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2005. 26

[BGP$^+$11] Lejla Batina, Benedikt Gierlichs, Emmanuel Prouff, Matthieu Rivain, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Mutual Information Analysis: a Comprehensive Study. *Journal of Cryptology*, 24(2):269–291, 2011. 24

[BN05] Yoojin Beak and Mi-Jung Noh. Differential Power attack and Masking Method. *Trends in Mathematics*, 8:1–15, 2005. 67, 96

[BNN$^+$12] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, and Georg Stütz. Threshold Implementations of All 3X3 and 4X4 S-Boxes. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2012. 19

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology, Crypto '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997. 5

[BSS$^+$13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013. 77

[CCD00] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential Power Analysis in the Presence of Hardware Countermeasures. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2000. 9

[CGP$^+$12a] Claude Carlet, Louis Goubin, Emmanuel Prouff, Michael Quisquater, and Matthieu Rivain. Higher-Order Masking Schemes for S-Boxes. In Anne Canteaut, editor, *Fast Software Encryption - FSE, 2012*, volume 7549 of *Lecture Notes in Computer Science*, pages 366–384. Springer, 2012. 28

[CGP$^+$12b] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. Conversion of Security Proofs from One Leakage Model to Another: A New Issue. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012 Proceedings*, volume 7275 of *Lecture Notes in Computer Science*, pages 69–81. Springer, 2012. 11, 85, 111

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure Conversion between Boolean and Arithmetic Masking of Any Order. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 188–205. Springer, 2014. xiii, xv, 10, 37, 69, 72, 73, 74, 98, 99

[CGVT15] Jean-Sébastien Coron, Johann Großschädl, Praveen Kumar Vadnala, and Mehdi Tibouchi. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In Gregor Leander, editor, *Fast Software Encryption- FSE 2015 - 22nd International Workshop Proceedings, To be Published*, Lecture Notes in Computer Science. Springer, 2015. 11, 57, 81

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Michael Wiener, editor, *Advances in Cryptology - CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999. 13, 14, 16, 17, 18, 38

[CLW06] Giovanni Di Crescenzo, Richard J. Lipton, and Shabsi Walfish. Perfectly Secure Password Protocols in the Bounded Retrieval Model. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography Conference, TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 225–244. Springer, 2006. 38

[Cor14] Jean-Sébastien Coron. Higher Order Masking of Look-Up Tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 441–458. Springer, 2014. 28, 41, 51, 110

[CPR07] Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. Side Channel Cryptanalysis of a Higher Order Masking Scheme. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 28–44. Springer, 2007. 28

[CPRR13] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-Order Side Channel Security and Mask Refreshing. In Shiho Moriai, editor, *Fast Software Encryption*, Lecture Notes in Computer Science, pages 410–424. Springer, 2013. 101, 107, 109, 110

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In Burton S. Kaliski, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002,* *4th International Workshop, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002. 8

[CRRY04] Scott Contini, Ronald L. Rivest, Matthew J. B. Robshaw, and Yiqun Lisa Yin. Improved Analysis of Some Simplified Variants of RC6. In Lars Knudsen, editor, *Fast Sowtare Encryption, FSE*, volume 1636 of *Lecture Notes in Computer Science*. Springer, 2004. 16

[CRV14] Jean-Sébastien Coron, Arnab Roy, and Srinivas Vivek. Fast Evaluation of Polynomials over Binary Finite Fields and Application to Side-Channel Countermeasures. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 170–187. Springer, 2014. 28

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. A New Algorithm for Switching from Arithmetic to Boolean Masking. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES, 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 89–97. Springer, 2003. 15, 32

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying Leakage Models: From Probing Attacks to Noisy Leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 423–440. Springer, 2014. 19

[Deb12] Blandine Debraize. Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2012. 15, 32, 34, 35, 71, 96

[DFS15] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. Making Masking Security Proofs Concrete - Or How to Evaluate the Security of Any Leaking Device. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 401–429. Springer, 2015. 19

[DPRS11] Julien Doget, Emmanuel Prouff, Matthieu Rivain, and François-Xavier Standaert. Univariate side channel attacks and leakage modeling. *Journal of Cryptographic Engineering*, 1(2):123–144, 2011. 112

[DSV$^+$14] François Durvaux, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Jean-Baptiste Mairy, and Yves Deville. Efficient Selection of Time Samples for Higher-Order DPA with Projection Pursuits. *IACR Cryptology ePrint Archive*, 2014:412, 2014. 23

[Dzi06] Stefan Dziembowski. Intrusion-Resilience Via the Bounded-Storage Model. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography Conference, TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 207–224. Springer, 2006. 38

[FIP01] Fips pub 197:, Specification for the Advanced Encryption Standard, National Institute of Standards and Technology, U.S. Department of Commerce, 2001. 2

[FRR$^+$10] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting Circuits from Leakage: the Computationally-Bounded and Noisy Cases. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 135–156. Springer, 2010. 18

[GBPV10] Benedikt Gierlichs, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. Revisiting Higher-Order DPA Attacks:Multivariate Mutual Information Analysis. In Josef Pieprzyk, editor, *Topics in Cryptology - CT-RSA 2010 Proceedings*, volume 5985 of *Lecture Notes in Computer Science*, pages 221–234. Springer, 2010. 24

[GBTP08] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual Information Analysis. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop 2008 Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 426–442. Springer, 2008. 8

[GJJR11] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A testing methodology for side channel resistance validation. In *NIST non-invasive attack testing workshop*, 2011. 25

[Gou01] Louis Goubin. A Sound Method for Switching between Boolean and Arithmetic Masking. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001. 15, 28, 29, 30, 39

[GPQ10] Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Secure Multiplicative Masking of Power Functions. In Jianying Zhou and Moti Yung, editors, *Applied Cryptography and Network Security, 8th International Conference, ACNS 2010 Proceedings*, volume 6123 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2010. 15

[GPQ11a] Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Montgomery's Trick and Fast Implementation of Masked AES. In Abderrahmane Nitaj and David Pointcheval, editors, *Progress in Cryptology - AFRICACRYPT 2011 - 4th International Conference on Cryptology in*

*Africa Proceedings*, volume 6737 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2011. 15

[GPQ11b] Laurie Genelle, Emmanuel Prouff, and Michaël Quisquater. Thwarting Higher-Order Side Channel Analysis with Additive and Multiplicative Maskings. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2011. 15

[Hey98] Howard Heys. A Timing Attack on RC5. In Stafford Tavares and Henk Meijer, editors, *Selected Areas in Cryptography , SAC 1998*, volume 1556 of *Lecture Notes in Computer Science*, pages 330–343. Springer, 1998. 5

[HOM06] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In *Applied Cryptography and Network Security, Second International Conference, ACNS 2006, volume 3989 of Lecture Notes in Computer Science*, pages 239–252. Springer, 2006. 26

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003. 15, 18, 28, 38, 39, 41, 42, 43, 50, 65, 69, 102

[JPS05] Marc Joye, Pascal Paillier, and Berry Schoenmakers. On Second-Order Differential Power Analysis. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005. 10

[KHL11] HeeSeok Kim, Seokhie Hong, and Jongin Lim. A Fast and Provably Secure Higher-Order Masking of AES S-Box. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2011. 11, 101, 104, 105, 107, 110, 126

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO 1999, 19th Annual International Cryptology Conference Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999. 8

[KJJR11] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to Differential Power Analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011. 6

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO 1996 Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. 5, 6

[KRJ04] Mohamed Karroumi, Benjamin Richard, and Marc Joye. Addition with Blinded Operands. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design, COSADE 2012 Proceedings*, volume 8622 of *Lecture Notes in Computer Science*. Springer, 2004. xiii, xv, 31, 67, 70, 72, 73, 74, 77, 78, 96, 98, 99

[KS73] Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *Computers, IEEE Transactions on*, 100(8):786–793, 1973. 58, 60, 61

[LM90] Xuejia Lai and James L. Massey. A Proposal for a New Block Encryption Standard. In Ivan Bjerre Damgård, editor, *Advances in Cryptology - EUROCRYPT 1990*, volume 473 of *Lecture Notes in Computer Science*, pages 389–404. Springer, 1990. 16

[MDS02] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Examining Smart-Card Security under the Threat of Power Analysis Attacks. *IEEE Trans. Computers*, 51(5):541–552, 2002. 21

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007. 5, 6

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-Channel Leakage of Masked CMOS Gates. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005. 19

[MPL$^+$11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2011. 19

[MR04] Silvio Micali and Leonid Reyzin. Physically Observable Cryptography (Extended Abstract). In Moni Naor, editor, *Theory of Cryptography Conference, TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2004. 38

[NIS95] NIST. Secure hash standard. In *Federal Information Processing Standard, FIPA-180-1*, 1995. 16, 71

[NP04] Olaf Neiße and Jürgen Pulkus. Switching Blindings with a View Towards IDEA. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 230–239, 2004. 32

[NW97] Roger M. Needham and David J. Wheeler. TEA Extentions. In *Technical report, Computer Laboratory, University of Cambridge*, 1997. 71

[OMHT06] Elisabeth Oswald, Stefan Mangard, Christoph Herbst, and Stefan Tillich. Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2006. 14, 23

[OMPR05] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A Side-Channel Analysis Resistant Description of the AES S-Box. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption*, volume 3557 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2005. 26

[PHL09] J. Pan, J. I. Hartog, and Jiqiang Lu. You Cannot Hide behind the Mask: Power Analysis on a Provably Secure S-Box Implementation. In Heung Youl Youm and Moti Yung, editors, *Information Security Applications, 10th International Workshop, WISA 2009, Busan, Korea, August 25-27, 2009, Revised Selected Papers*, pages 178–192. Springer-Verlag, Berlin, Heidelberg, 2009. 85

[PR] Emmanuel Prouff and Matthieu Rivain. Masking against Side-Channel Attacks: A Formal Security Proof. In *Advances in Cryptology - EUROCRYPT 2013*. 19

[PRB09a] E. Prouff, M. Rivain, and R. Bevan. Statistical Analysis of Second Order Differential Power Analysis. *Computers, IEEE Transactions on*, 58(6):799–811, June 2009. 120, 121

[PRB09b] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. Statistical Analysis of Second Order Differential Power Analysis. *IEEE Trans. Computers*, 58(6):799–811, 2009. 24

[PSQ07] Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Power and Electromagnetic Analysis: Improved Model, Consequences and Comparisons. *Integr. VLSI J.*, 40(1):52–60, January 2007. 112

[RDJ$^+$01] Atri Rudra, PradeepK. Dubey, CharanjitS. Jutla, Vijay Kumar, JosyulaR. Rao, and Pankaj Rohatgi. Efficient Rijndael Encryption Implementation with Composite Field Arithmetic. In ÇetinK. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2001. 104

[RDP08] Matthieu Rivain, Emmanuelle Dottax, and Emmanuel Prouff. Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis. In Kaisa Nyberg, editor, *Fast Software Encryption, FSE 2008 Proceedings*, volume 5086 of *Lecture Notes in Computer Science*, pages 127–143. Springer, 2008. 26, 27, 39, 82, 84, 85, 86, 90, 91, 92, 93, 112, 113, 114, 126

[RGV12] Oscar Reparaz, Benedikt Gierlichs, and Ingrid Verbauwhede. Selecting Time Samples for Multivariate DPA Attacks. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2012. 23

[Rot12] Guy N. Rothblum. How to Compute under ${\cal{AC}}^{\sf0}$ Leakage without Secure Hardware. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 552–569. Springer, 2012. 18

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably Secure Higher-Order Masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010. 28, 44, 51, 101, 103, 104, 106, 110

[RPD09] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 171–188. Springer, 2009. 9

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. 3

[RV13] Arnab Roy and Srinivas Vivek. Analysis and Improvement of the Generic Higher-Order Masking Scheme of FSE 2012. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 417–434. Springer, 2013. 28

[Sko05] Sergei P. Skorobogatov. Semi-invasive attacks – A new approach to hardware security analysis, PhD Thesis, 2005. 5

[SLP05] Werner Schindler, Kerstin Lemke, and Christof Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2005. 8, 112

[SMTM01] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In Colin Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2001. 104, 105

[SMY09] François-Xavier Standaert, Tal Malkin, and Moti Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009 Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 443–461. Springer, 2009. 18

[SP06] Kai Schramm and Christof Paar. Higher Order Masking of the AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006. 27

[Sta77] Data Encryption Standard. FIPS pub 46. *Appendix A, Federal Information Processing Standards Publication*, 1977. 2

[SVO$^+$10] François-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. The World Is Not Enough: Another Look on Second-Order DPA. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 112–129. Springer, 2010. 18

[TWO13] Michael Tunstall, Carolyn Whitnall, and Elisabeth Oswald. Masking Tables - An Underestimated Security Risk. In Shiho Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 425–444. Springer, 2013. 85

[VG13] Praveen Kumar Vadnala and Johann Großschädl. Algorithms for Switching between Boolean and Arithmetic Masking of Second Order. In Benedikt Gierlichs, Sylvain Guilley, and Debdeep Mukhopadhyay, editors, *Security, Privacy, and Applied Cryptography Engineering - Third International Conference, SPACE 2013 Proceedings*, volume 8204 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2013. 11, 90

[VG15] Praveen Kumar Vadnala and Johann Großschädl. Faster Mask Conversion with Lookup Tables. In Stefan Mangard and Axel Y. Poschmann, editors, *Constructive Side-Channel Analysis and Secure Design - Sixth International Workshop, COSADE 2015 Proceedings, To be published*, Lecture Notes in Computer Science. Springer, 2015. 11, 81

[VMKS12] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 740–757. Springer, 2012. 9

[VS09] Nicolas Veyrat-Charvillon and François-Xavier Standaert. Mutual Information Analysis: How, When and Why? In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 429–443. Springer, 2009. 8

[WVGX15] Junwei Wang, Praveen Kumar Vadnala, Johann Großschädl, and Qiuliang Xu. Higher-Order Masking in Practice: A Vector Implementation of Masked AES for ARM NEON. In Kaisa Nyberg, editor, *The Cryptographer's Track at the RSA Conference 2015 . Proceedings, To be Published*, Lecture Notes in Computer Science. Springer, 2015. 11, 101

# List of publications

1. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. *Will be published in the proceedings of FSE, 2015.* Co-authored with Jean-Sébastien Coron, Johann Großschädl and Mehdi Tibouchi.

2. Faster Mask Conversion with Lookup Tables. *Will be published in the proceedings of COSADE, 2015.* Co-authored with Johann Großschädl.

3. Higher-Order Masking in Practice: A Vector Implementation of Masked AES for ARM NEON *Published in the proceedings of CT-RSA, 2015.* Co-authored with Johann Großschädl, Junwei Wang and Qiuliang Xu.

4. Secure Conversion between Boolean and Arithmetic Masking of Any Order. *Published in the proceedings of CHES, 2014.* Co-authored with Jean-Sébastien Coron and Johann Großschädl.

5. Implementation and Evaluation of a Leakage-Resilient ElGamal Key Encapsulation Mechanism. *Published in the proceedings of Proofs 2014, Invited to the Journal of Cryptographic Engineering.* Co-authored with David Galindo, Johann Großschädl, Zhe Liu and Srinivas Vivek.

6. Algorithms for Switching between Boolean and Arithmetic Masking of Second Order. *Published in the proceedings of SPACE, 2013.* Co-authored with Johann Großschädl.

7. Conversion of Security Proofs from One Model to Another: A New Issue. *Published in the proceedings of COSADE, 2012.* Co-authored with Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner and Matthieu Rivain.

8. Full Key Recovery Attacks on Modular Addition: An Application to Threefish. *Published in the proceedings of WESS, 2012.* Co-authored with Jean-François Gallais and Arnab Roy.