Efficient Side-channel Resistant MPC-based Software Implementation of the AES

Abraham Fernandez Rubio
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/etd-theses

Repository Citation
https://digitalcommons.wpi.edu/etd-theses/403
Efficient Side-channel Resistant MPC-based Software Implementation of the AES

by

Abraham Fernandez-Rubio

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical Computer Engineering

by

April 2017

APPROVED:

Professor Thomas Eisenbarth, Thesis Advisor

Professor Xinming Huang, Thesis Committee

Professor Yehia Massoud, Head of Department
Abstract

Current cryptographic algorithms pose high standards of security yet they are susceptible to side-channel analysis (SCA). When it comes to implementation, the hardness of cryptography dangles on the weak link of side-channel information leakage. The widely adopted AES encryption algorithm, and others, can be easily broken when they are implemented without any resistance to SCA. This work applies state of the art techniques, namely Secret Sharing and Secure Multiparty Computation (SMC), on AES-128 encryption as a countermeasure to those attacks. This embedded C implementation explores multiple time-memory trade-offs for the design of its fundamental components, SMC and field arithmetic, to meet a variety of execution and storage demands. The performance and leakage assessment of this implementation for an ARM based micro-controller demonstrate the capabilities of masking schemes and prove their feasibility on embedded software.
In memory of my mother and father
Acknowledgments

This work represents the sum of all the support from all of my circles, my family, friends and fellow students. I feel grateful to contribute, even in the most insignificant way, to science and humanity.

I arrived at the right time to the WPI and had the great opportunity to be advised by professor Thomas Eisenbarth, a very talented and hard-working researcher. Beneath this work are countless moments of discussion, stress, learnings and laughter we shared. I thank him for all the patience, support and belief in me.

I thank professor Yehia Massoud for being part of this thesis committe and professor Xinming Huang for his professional opinion and his support as a committe member.

During these ten months of research, the Vernam lab has been my home in which I’ve been surrounded by great fellow lab members. I want to thank Okan Seker and Cong Chen for their collaboration on this research and their patience to share their knowledge with me.

I am very grateful with the programs that have sponsored my studies and stay in the US: the Fulbright program for opening the door for me to this country, CONACYT for their constant support, the WPI for their tremendous support and acceptance as a fellow, all of these institutions have allowed me to focus on my studies and embrace the research experience.

This thesis is based upon work supported by the National Science Foundation under Grant No.1261399.

Special thanks to Neda Seyedmahmoud for her support and review in the writing of this thesis. Also, Jessica O’Toole for her guidance in the writing process.

This work would have not been possible without the constant love and continuous support by Elisa Montano, her care and attention regardless of the distance have been fundamental.
## Contents

1 Introduction ................................. 1
   1.1 Motivation ...................................... 1
   1.2 Related Work ................................ 1
   1.3 Contribution ................................ 2
   1.4 Outline ..................................... 3

2 Background .................................... 4
   2.1 Cryptography in Embedded Devices ................. 4
   2.2 AES ........................................ 6
   2.3 Masking ..................................... 8
      2.3.1 Shamir’s Secret Sharing .................... 9
      2.3.2 Secure Multiparty Computation (SMC) ........... 10
   2.4 Side-Channel Analysis ........................... 11

3 Masking Implementation ......................... 14
   3.1 Specifications ................................ 14
   3.2 Performance .................................. 20
   3.3 Challenges .................................. 21

4 Leakage Assessment ............................... 23
   4.1 Setup ....................................... 23
   4.2 Challenges .................................. 24
      4.2.1 Trace Generation ........................... 24
      4.2.2 Trace Collection ........................... 25
      4.2.3 Trace Processing ........................... 25
   4.3 SCA Results .................................. 26
      4.3.1 Higher-Order t test ....................... 26
      4.3.2 Multivariate t test ....................... 30

5 Conclusion ...................................... 35

6 Appendix ....................................... 36

References ...................................... 37
1 Introduction

1.1 Motivation

In the current technological age, millions of devices are connected to the internet, yet the projected amount is expected to reach 200 billion by 2020 [Int]. From a smart watch attached to our hand to the Milkyway-2, one of the world’s fastest supercomputers [Alb], the computing demand is ubiquitous but the level of security in the Internet of Things (IoT) remains uncertain. Computer security has become a priority as our dependency on technology leads us to a point where we have to trust computers for most of our daily activities. Even though the computer industry has been incorporating secure algorithms in their hardware and software designs, there’s a facet of it that is frequently unattended known as side-channel analysis (SCA).

Side-channel analysis is a branch of computer security that focuses on gathering and correlating big amounts of data inherently related to the behavior of cryptographic algorithm implementations. The information derived from the management of resources like execution time [Koc96], energy [KJJ99] [GMO01, QS01] and memory of an electronic device reveals valuable clues about the internal actions it performs. Single samples of data cannot tell much about the internal state of the algorithm but in the aggregate, after collecting thousands or millions of traces, an observer would have enough information to make educated guesses regarding the secrets processed by the device. The objective of this analysis is to either assess the level of security offered by the implementation or break into it by learning from the leaked information about the internal state of the algorithm.

Encouraged by the current trend of determining the efficiency of practical cryptographic implementations and their leakage assessment, this thesis inspects the level of resistance against side-channel analysis of a software implementation of the widely-used Advanced Encryption Standard (AES). This version of AES is based on [SES17] that prevents side-channel leakage by incorporating state of the art masking techniques and multi-party computation of the secrets processed by the algorithm. The concept behind a masking scheme is to split a secret into multiple pieces called shares. Each share is independent and can be processed individually without revealing any information about the secret and in the end, all the shares are combined to yield the processed secret.

The characterization of a masking scheme is defined by two parameters: the number of random shares \( n \) and the smallest possible number \( d + 1 \) required to reconstruct the secret variable. The order of the masking scheme is determined by the magnitude of \( d \), thus an \((n, d)\)-sharing scheme does not reveal any information under a \( d^{th}\)-order side-channel analysis. Due to the exponential proportion between the order and the complexity of mounting an attack on a certain sharing scheme, it brings acceptable results as \( d \) grows [CJRR99].

1.2 Related Work

Kocher et al. [KJJR11] describe in detail the characteristics of Differential Power Analysys (DPA) and Simple Power Analysis (SPA). These methods are capable of learning valuable information about secrets processed by cryptographic algorithm implementations when they are not properly...
protected against side-channel attacks. In the presence of enough leakage it is possible to recover the keys. The work by [SM15] points out the current leakage assessment methodologies, these tests are useful for the designer to understand the side-channel behavior of the implementation and reveal potential sources of leakage if any.

Security researchers have been proposing a wide variety of countermeasures to defend cryptographic algorithm implementations from SCA. Rivain and Prouff [RP10] presented the first generic \( d \)-order masking scheme for AES with provable security and acceptable software implementation overhead. That contribution is based on Ishai’s proposal [ISW03]. Their work extends the boolean masking schemes to any finite field and by incorporating provable secure operations on masked variables they achieve an efficient software AES implementation. According to it, an \((n, d)\) masking scheme with \( n = d + 1 \) allows \( d^{th}\)-order level of security but Coron et al. [CPRR14] indicated that it requires \( n \geq 2d + 1 \) shares.

Another countermeasure against passive SCA, introduced by von Willich [vW01], presents a new technique known as affine masking. It suggests an increase in the number of intermediate variables required to recover any secret information, it also changes the variables randomly on every run. Affine masking can achieve similar performance to Boolean masking, however it has not been generalized for orders \( d > 1 \).

One of the first influential concepts was brought by Shamir [Sha79] where a randomly generated degree \( d \) polynomial is used to break a confidential variable into \( n \) parts without revealing any information about it. The secret can be reconstructed from the \( n \) shares but an adversary cannot recover any information from \( d \) number of them, it is commonly assumed that \( n = d + 1 \).

Secure multi-party computations were introduced initially by Ben-Or et al. [BOGW88] and Gennaro et al. [GRR98], they represent the important counterpart of secret sharing. Later the work by Goubin [GM11], Roche and Prouff [RP12] exploit polynomial masking. They demonstrate that their construction thwarts \( d^{th}\)-order side-channel attacks even in the presence of glitches.

In summary, it is important to identify the difference between masking schemes and secure multi-party computation. The former describes how to break a secret into multiple shares and the latter carries out the processing of the parts preserving the completeness of the secret. The purpose of this layer of redundancy is to keep the confidentiality of the sensitive variables from side-channel leakage. The downside of SCA resistant implementation is the computation performance overhead inherently derived from the need to process the cryptographic algorithms for every share. Grosso et al. [GSF14] evaluate the performance cost introduced by higher-order masking schemes for AES, and additional security features like glitch-freeness require an elevated computation overhead.

1.3 Contribution

This thesis extends the work [SES17] in which a polynomial masking and secure multi-party computation [RP12] approach is applied in software to harden the widely used AES. My work provides higher-order side-channel analysis (HO-SCA) results, a prototype implementation released to the public \(^1\) and in-depth performance analysis. The leakage assessment and performance analysis are done on the ultra-low power ARM Cortex-M0+. This code is an 8-bit implementation written in C

\(^1\)https://github.com/vernamlab
and it can be easily ported to any platform, it also provides 5 different \((n, d)\) secret sharing schemes that are selectable by pre-compiler parameters, however the code is easily expandable to higher orders. Most of the contributions by the side-channel research community have been focused on hardware implementations [MM13, DCBRN15]. Although recently Goudarzi and Rivain [GR16] thoroughly analyzed the performance of higher-order masking techniques in software for ARM architecture, but their work lacks leakage assessment.

The leakage assessment included in this thesis comprises univariate \(t\) tests and second-order multivariate \(t\) tests. These tests are capable of highlighting the relationship between the processed sensitive variables and side-channel leakage. Due to the long execution times of the masking scheme, the HO-SCA is focused on the secure multiplication function that is fundamental to the algorithm. The results show that, statistically, the power consumption is independent of the processed secrets.

The implementation features several execution optimizations, according to the target platform memory and execution limitations the user can alternate between different versions of the field multiplications. Additionally, since the code is written in a branchless fashion to prevent microarchitectural leakage, all the available versions of the functions execute in constant time. Performance measurements for all version of secure multiplication for different \((n, d)\)-sharing schemes are also discussed. A breakdown of the operation per round of AES-128 and execution timings are also given. Finally, a table comparing code and memory sizes is provided.

1.4 Outline

This thesis is organized in three main sections: general background, implementation details, performance results and leakage assessment. Section 2 provides a brief description of main concepts such as AES-128, an introduction to polynomial masking, secure multiparty computation and side-channel leakage assessment methodologies. Section 3 describes the considerations and low-level details of bringing the scheme to the ARM based microcontroller as well as performance results. Finally, Section 4 shows a comprehensive side-channel analysis for \((3,1)\)-sharing and \((5,2)\)-sharing schemes including the challenges to perform the assessment and its results.
2 Background

There are important considerations in the applied cryptography and security industry that must be satisfied to reduce the surface of attack of a product or service. Current cryptographic algorithms have been closely scrutinized by the cryptography research community before these algorithms were globally adopted. These selections are based on a history of learnings from broken crypto-algorithms, new contributions to cryptoanalysis and the current computational power pushing the boundaries of brute-force attacks. Yet, once these algorithms are transitioned from the theory to their practical implementation, new entry points emerge like side-channel analysis (Section 2.4) and fault injections. One of reasons is that crypto-algorithms have to be secure but they have to be efficient as well, some of them perform better in software and others in hardware; but to provide another layer of protection against SCA and physical attacks, certain level of redundancy or extra processing effort is required. Unfortunately, it is very common to find implementations that prioritize the user experience at the cost of sacrificing security. So, it is fundamental to understand the market segments and the conditions where secure devices will be operating in order to find the appropriate trade-off between security and performance.

2.1 Cryptography in Embedded Devices

Today, embedded devices are a key part of our lives from the big servers processing financial transactions to the small ones inside smartphones. The omnipresence of these tiny general purpose computers, that handle specific tasks, process big part of the information that we need everyday. Embedded systems are virtually everywhere, from the industry like machinery and tools, inside our homes like in appliances and air conditioners, transportation like cars and planes, in space like satellites and spaceships, as well as part of our gadgets like phones, tablets and credit cards.

This section classifies the handling of sensitive information into three high-level stages: communication, processing and storage. To illustrate this process assume a user, called Alice, who wants to log in to her home PC. She has to introduce her credentials (user ID and password) and then make sure nobody is watching as she types the information. In other words, the communication channels have to be secure including the keyboard that is used to input the secret data. Later, the computer checks the credentials to verify her identity. Ideally, the processing of the data should not reveal any information to an adversary looking at side-channel effects like cache usage, power consumption or execution time. Finally, the password must be stored in a way that even if an attacker breaks into the computer, the attacker should not be able to get the password. In other words, the password must be scrambled in a deterministic way from which the password can be verified but not guessed.

Communication. The embedded electronics sector has been tremendously pushed by the telecommunications industry. It is not only the necessity of human beings to communicate with each others but a fundamental requirement to transfer a bit of information from one place to another. Thus, our communicated world heavily depends on moving data around in a reliable and practical way, but when the sensitivity of the information being transferred demands it, security plays a big role. Think of the most common actions that people carry out every day like sending
money from our bank account to another part of the world using our phone, paying for a cup of
coffee with a credit card, making a phone call or simply just posting a message on social media;
these large scale examples involve dealing with the exchange of sensitive information.

Processing. Once the sensitive data is inside a computing core, it follows a series of instructions
to operate on the data. For example, arithmetic, logical, flow control operations, etc. It is im-
portant to have the protection mechanisms inside the core to prevent an adversary from accessing
restricted information. For instance, access to memory should be limited to certain range, impor-
tant configuration registers should have locking mechanisms, the core should support hierarchical
levels of execution, and many more complex protections. Even if these considerations are put in
place, the execution time and power consumption of the instructions can reveal certain information
about the data being processed. The reason is that the data representation within the core and
the way it is handled relates to side-channel variables.

Storage. Inevitably, the information has to be saved temporarily, even just for a few nano-
seconds like in a cache memory or for longer periods in a hard drive. The confidentiality/privacy
level of the data should match its level of protection. For example, if the information is sensible
and is stored in clear text then it should have restricted access, or if it requires stronger protection
then it should be encrypted. There have been recent attacks that exploit the cache memory access
time as a source of leakage to recover cryptographic keys [BM06] and [IIES14].

A well designed implementation of a cryptographic algorithm should systematically consider
these three important stages at all levels. In other words, security can be applied to all the
layers that a system is composed of, either at the level of a System-on-Chip (SoC) or at the
level of a wireless telephone network or any other. There are many cryptographic primitives
that can be used to guarantee the security objectives of an embedded system. For example,
symmetric and asymmetric encryption algorithms to provide confidentiality, cryptographic hashes
to assess integrity and signature schemes for authentication and non-repudiation. As the security
of embedded devices directly depends on the strength of cryptographic algorithms, it also depends
on the resistance of the implementation to side-channel attacks.

Many embedded devices feature cryptographic primitives as part of their design. Either as part
of their hardware modules or as an extension of the set of instructions that they support, these
features can improve the performance of a secure application. Hardware cryptographic modules or
extensions tend to be faster as they are dedicated machines, however the cost and size of the die
is also higher compared to software implementations. The demand of either software or hardware
assisted cryptography relies on the market segment and usage model of the application. For
example, if the user has physical possession of the device then she/he could manage to perform a
wider range of attacks as oppose to having remote access. There are many other considerations like
the user access privilege and the connectivity of the application in order to patch any vulnerabilities
remotely.

As the presence of embedded devices is very wide, product developers and security design-
ers must consider an increasing amount of attack sources and possible protections against them.
This thesis focuses on the design, performance and side-channel assessment of a robust software
implementation of the AES encryption algorithm in an embedded microcontroller.
2.2 AES

The Advanced Encryption Standard (AES) was built on top of three pillars [DR99, p. 8]: simplicity, efficiency and resistance. The algorithm, initially called Rijndael named after its creators, is compact in terms of code size and its simplicity allows performance efficient implementations. Although the algorithm itself was designed to resist all known attacks by then, most current implementations are still subject to side-channel attacks.

AES is a block cipher that supports a block size of 128 bits and key lengths of 128, 192 or 256 bits. The algorithm operates iteratively on the intermediate mixture of plaintext and key called state. The number of iterations (rounds) depends on the block and key sizes, it can be 10, 12 or 14. For simplicity and to focus on the particular AES implementation of this thesis, the rest of this section describes the AES-128 that works on 128-bit key, a 128-bit block and 10 rounds in ECB (Electronic Codebook) mode. Further details about the other block/key sizes and number of rounds can be found in [DR99, p. 9]. In AES-128, the state and the key can be pictured as two different arrays of 4-by-4 bytes, the first four bytes compose the first column, byte 0 at the top and byte 3 at the bottom. Next column from left to right represents the next four bytes and so on. This representation simplifies the explanation of how the functions operate on the state and keys.

The algorithm is composed of four invertible main functions: \textit{SubBytes}, \textit{ShiftRows}, \textit{MixColumns} and \textit{AddRoundKey}. For the first nine rounds, all functions are called but in the last round all excepting \textit{MixColumn} are used. The next diagram illustrates how these functions are structured for encryption.

![AES-128 state and key represented as arrays of 16 Bytes arranged in column major order.](image)

The algorithm is composed of four invertible main functions: \textit{SubBytes}, \textit{ShiftRows}, \textit{MixColumns} and \textit{AddRoundKey}. For the first nine rounds, all functions are called but in the last round all excepting \textit{MixColumn} are used. The next diagram illustrates how these functions are structured for encryption.
Figure 2: AES-128 Encryption high-level diagram. Plaintex and key are the inputs, ciphertext is the output. The blocks in purple represent the functions that are applied to the intermediate state on every round.

The *SubBytes* is simply an arbitrary substitution of the state bytes with another set of bytes based on a look-up table called S-Box, this function is a non-linear substitution, it basically means that it cannot be replaced by boolean operations AND and XOR. The S-box can be expressed as two operations on each byte of the state: multiplicative inverse and then an affine transformation, both over GF($2^8$). To invert the process, first the inverse of the affine mapping is applied and then the multiplicative inverse.

The *ShiftRows* transformation rotates to the left the bytes within each column of the state, the first row is left without rotation, the second row is rotated by one position, the third row is rotated by two positions and the fourth by three positions. To invert the process, the first row is again left without rotation, the second, third and fourth rows are rotated to the right by one, two and three bytes respectively.
In the MixColumns function, every column of the state is multiplied by an invertible polynomial $b(x) = 03x^3 + x^2 + x + 02$ modulo $x^4 + 1$ over GF($2^8$). To reverse this process, the inverse polynomial $c(x) = 06x^3 + 0dx^2 + 09x + 0e$ is multiplied by all the columns of the state.

During the AddRoundKey, each byte of the expanded key is mixed with its corresponding byte of the state by an XOR operation. It is the simplest function but for every round the expanded key is different. In order to get a different expanded key for every round, there’s a key schedule mechanism. It takes the previous key and other operations like rotate, S-box transformation, and one exponentiation of 2 over GF($2^8$) for every round. A full description of the key expansion and key selection algorithm can be found in [DR99, pp.14,15].

### 2.3 Masking

An adversary can perform a side-channel analysis on a device running a cryptographic algorithm and the complexity of the attack can be extended to different points in time and different sources of information. This type of physical attack is considered a Higher-Order Side-Channel Analysis (HO-SCA), the number of different points of observation (leakages) determines the order of the
attack [RP12, p.111]. There is an exponential relationship between the order and the complexity
of such attack. But it is important to mention that not only the attack becomes more complex,
also a cryptographic implementation resistant to HO-SCA requires more computing resources to
successfully thwart the analysis.

Masking, also known as sharing, is a technique to prevent HO-SCA [CJRR99, pp.146,147] [RP12, pp.111,114]. It basically consists in splitting a sensitive variable into multiple parts, all these shares hold a deterministic relationship to the sensitive variable but the processing of these individual parts may not reveal enough information to break the cryptographic algorithm under test. A sensitive variable depends on parameters of the algorithm that are secret, for example plaintext mixed with keys as in the AES intermediate state.

2.3.1 Shamir’s Secret Sharing

The power consumption due to the processing of a sensitive variable by a cryptographic device
may reveal valuable information about the variable itself. The reason of that leakage is because
there is a relationship between the variable, the instructions operating on it and the side-channel
information coming out of the device, e.g. the power consumption. An adversary, collecting a
certain amount (say a few thousands) of power traces, would be able recover the variable by doing
some statistical analysis that will be described later in section 2.4. Adi Shamir in his [Sha79]
introduces Shamir’s Secret Sharing a very clever concept that [RP12] use as a countermeasure to
help thwart this attack.

Shamir’s Secret Sharing splits a sensitive variable $s$ into multiple pieces that are related to
the secret by a degree-$d$ polynomial $P(x) = s + a_1 x + \ldots + a_d x^d$ where the coefficients $a_i$’s are
randomly selected and are meant to remain secret throughout the lifetime of the share. A number
$n$ of parties $(I_i)_{i=0,\ldots,n-1}$ get a new share by evaluating the polynomial at different non-zero points
$\alpha_0,\ldots,\alpha_{n-1}$ that are public, the resulting points $P(\alpha_0),\ldots,P(\alpha_{n-1})$ are known as shares. The
number of shares $n$ and the degree $d$ of the polynomial specify the characteristics of the masking
scheme, in other words, $(n,d)$-sharing. The relationship between the number of shares and the
order of the polynomial can be $n = d + 1$ [RP12, p.111] for linear operations among the secret
shares. This is because in order to reconstruct the coefficients of the polynomial, $d + 1$ points are
required. Figure 1 describes the masking scheme in a matrix equation representation.

$$
\begin{bmatrix}
P(\alpha_0) \\
P(\alpha_1) \\
\vdots \\
P(\alpha_d) \\
P(\alpha_{n-1})
\end{bmatrix} =
\begin{bmatrix}
1 & \alpha_0 & \alpha_0^2 & \ldots & \alpha_0^{n-1} \\
1 & \alpha_1 & \alpha_1^2 & \ldots & \alpha_1^{n-1} \\
\vdots \\
1 & \alpha_d & \alpha_d^2 & \ldots & \alpha_d^{n-1} \\
\vdots \\
1 & \alpha_{n-1} & \alpha_{n-1}^2 & \ldots & \alpha_{n-1}^{n-1}
\end{bmatrix}
\begin{bmatrix}
s \\
a_1 \\
a_2 \\
\vdots \\
a_d \\
a_{n-1}
\end{bmatrix}
$$

From the equation (1) above, the matrix composed of the public points $(\alpha_i^j)_{i,j=0,\ldots,n-1}$ is known
as the Vandermonde matrix $V$, it is a square matrix of $n \times n$. The coefficients $(a_i)_{i=1,\ldots,n-1}$ are
randomly generated by the masking scheme and the user does not need to have access to them. It is important to note that to reconstruct the secret value $s$, only the first row (expressed as $\lambda_0, \ldots, \lambda_{n-1}$) of the inverse Vandermonde matrix is required such that $s = \sum_{i=0}^{n-1} P(\alpha_i)\lambda_i$.

### 2.3.2 Secure Multiparty Computation (SMC)

Once secret sharing has been introduced, the next step is how to take advantage of its level of confidentiality to establish a suitable side-channel analysis resistant masking scheme. From the previous section, a user can mask a secret by splitting it into multiple parts so no information can be extracted from each individual share or the accumulation of $d$ shares together. The inverse is also possible, the user is able to reconstruct the secret by polynomial interpolation with the knowledge of $d + 1$ shares. To make use of this elegant mechanism, the user also needs to know how to do computations on the shares [BOGW88] such that they cause the same effect on the secret value. This section briefly describes three fundamental operations required to implement AES encryption, the operations act on the secret shares but produce the same result as if the secret variable is operated directly.

To compute linear operations based on shares of a secret variable the process is straightforward. Consider a fixed finite field $E$ and $s_1, s_2$ are two secret values that belong to $E$ and have been previously generated by two different polynomials $f(x)$ and $g(x)$ respectively. Also consider two non-zero constants $c_1$ and $c_2$ that belong to the same field $E$. The addition of shares $h(\alpha_i) = f(\alpha_i) + g(\alpha_i)$ of all parties $i = 0, \ldots, n - 1$ return the same result as if the two secrets are added together $s_1 + s_2$. In the same way, the affine transformation $k(\alpha_i) = c_1 \cdot f(\alpha_i) + c_2$ of the shares of all parties correspond to the affine transformation of the secret $c_1 \cdot s_1 + c_2$.

However, the multiplication of two masked secrets is a non-linear operation that involves a more complicated process. After its introduction in [BOGW88, p.4], the secure multiplication of two masked secrets was later improved by [GRR98] and afterwards by [RP12, p.119], the high-level description of the algorithm is defined as follows:

1. Each player $I_i$ computes $h(\alpha_i) = f(\alpha_i) \cdot g(\alpha_i)$ locally,
2. $I_i$ generates a degree $d$ polynomial $Q_i(x)$ such that, $Q_i(0) = h(\alpha_i)$ and sends the value $Q_i(\alpha_j)$ to player $I_j$.
3. $I_i$ computes $Q(\alpha_i) = \sum_{j=0}^{n-1} \lambda_j Q_j(\alpha_i)$ where $(\lambda_0, \ldots, \lambda_{n-1})$ represents the first row of the inverse Vandermonde matrix.
4. The family $Q(\alpha_i)_{i=0,...,n-1}$ is a shared representation of $s_1 \cdot s_2$.

Based on this algorithm, equations (2) and (3) show an example of the matrix representation of the operations necessary for a secure multiplication of two shared secrets in the $(5,2)$-masking scheme. Note that these equations only summarize the required field multiplications and additions but not how they are processed by the different players.
\[
Q = \begin{bmatrix}
    f(\alpha_0) \cdot g(\alpha_0) & r_{0,0} & r_{0,1} \\
    f(\alpha_1) \cdot g(\alpha_1) & r_{1,0} & r_{1,1} \\
    \vdots & \vdots & \vdots \\
    f(\alpha_4) \cdot g(\alpha_4) & r_{4,0} & r_{4,1}
\end{bmatrix}
\begin{bmatrix}
    \alpha_0^0 & \alpha_1^0 & \ldots & \alpha_4^0 \\
    \alpha_0^1 & \alpha_1^1 & \ldots & \alpha_4^1 \\
    \alpha_0^2 & \alpha_1^2 & \ldots & \alpha_4^2
\end{bmatrix}
\]

(2)

\[
Q = \begin{bmatrix}
    \lambda_0 & \lambda_1 & \ldots & \lambda_4
\end{bmatrix} \cdot Q
\]

(3)

It is important to note that when two sets of shares derived from degree \(d\) polynomials are multiplied, the resulting degree would be \(2d\), then \(2d + 1\) points are required to reconstruct the shared secret. A particular case of the multiplication is the squaring operation, due to a property introduced [RP12, p.118], it can be simplified if pairs of public points denoted \(\alpha_i\) and \(\alpha_j\) that follow \(\alpha_i \neq 0\) for \(i = 1, \ldots, n\) and if for each \(\alpha_i\) there is an \(\alpha_j\) that satisfies \(\alpha_i^2 = \alpha_j\) then the transformation \(\eta_k(y) = c_1 \cdot y^{2^k}\) is possible.

Each player calculates the transformation on its share locally by \(s(\alpha_i) = c_1 \cdot [f(\alpha_i)]^{2^k} = f'(\alpha_j)\) where \(f'(x)\) is the polynomial whose coefficient are calculated by applying the transformation to the coefficients of \(f(x)\). The family of shares \(s(\alpha_i)\) for \(i = 0, \ldots, n-1\) is a valid set of secret shares of \(c_1 \cdot s_1^{2^k}\). However, communication between players is needed to do the reordering of the secret shares.

Notice that there is a performance cost that has to be paid in order to break every secret variable into shares and do computations on them. An embedded device running a masking scheme should use the appropriate trade-off between the level of resistance against side-channel attacks and the desired performance. Based on these considerations and the effort that the fundamental operations represent, [RP12] created an AES implementation resistant to side-channel attacks. Also recently, [GR16] proposed a fine-tuned AES and analyzed its performance. The corresponding details of the implementation of these and other operations are presented in section 3.1, their performance is described in section 3.2 and the SCA of the secure multiplication is shown in section 4.3.

2.4 Side-Channel Analysis

As electronic devices are more common part of our lives and they handle confidential and private data, the industry must provide high standards of security embedded into the devices. The side-channel research community has been developing more sophisticated attacks and the computing tools are becoming faster and more efficient. Storage is cheaper and more available than it was a few years ago so the analysis of side-channel data is becoming more accessible. Nowadays, the budget required to perform HO-SCA is reachable by low-profile adversaries, thus the electronics industry must agree on a standard set of tests and methodologies to assess the level of security against side-channel attacks on embedded devices [SM15, p.1]. This section describes the \(t\) test, one of the most common analyses to detect if a cryptographic implementation is leaking potentially valuable data.

Proposed by Gilbert et al. [GGJR+11], and later used in [BGN+14] and [LMW14], the \(t\) test reveals if there’s any side-channel leakage in a device-under-test (DUT) i.e. a device running a
A cryptographic algorithm. Note that the test itself is not an attack on the DUT, it cannot recover the secret keys but it helps to determine whether it is possible to perform more specific attacks to exploit the leakage and retrieve sensitive data. This test is based on the concept that the implementation is potentially free of leakage if two side-channel sets of measurements, e.g. power traces taken under two different situations, are indistinguishable from one another. In other words, there should not be a correlation between two (usually large) sets of data and the conditions under which the sets were captured.

The t test has an statistical foundation, as [SM15, p.2] very precisely point out: "a fundamental question in many different scientific fields is whether two sets of data are significantly different from each other". This test quantifies the probability of the distinction between the averages of two sets of data. The measurements can be collected in a variety of different ways but to get relevant results and to prevent false-positives it is important to take some considerations, like the way the measurements are catalogued and the order in which they are extracted from the embedded device. The analysis in this thesis uses the non-specific t test, it consists on taking all the samples in a random fashion based on a random pattern that must be appropriately logged to later process the samples accordingly. The DUT processes a random input or a fixed input based on the value of the random pattern, for example if a selected bit of the pattern is 1 then the input is selected to be random or fixed if the opposite and so on for the rest of the bits in the pattern. This prevents false-positive dectections since the state of the device is being exercised under the same thermal and electrical conditions, otherwise the difference in these parameters could bias the result of the test.

After collecting and categorizing the traces according to the logged pattern, the means ($\mu_f$, $\mu_r$) and the variances ($\sigma_f^2$, $\sigma_r^2$) for the two sets are calculated. Finally, the Welch’s t test is executed as in Equation 4 where $n_f$ and $n_r$ denote the cardinality of the fixed and random sets respectively.

$$t = \frac{\mu_f - \mu_r}{\sqrt{(\sigma_f^2/n_f) + (\sigma_r^2/n_r)}}.$$  \hspace{1cm} (4)

Notice that the performance of the calculation will depend on the amount of traces and their length. Also the number of random traces and the number of fixed traces would differ slightly since their occurrence is selected randomly. Notice that for every point in time there is a t value, thus this test is considered the univariate or first order t test. An (n,2)-masked implementation should not reveal leakage under this test, for an appropriate selection of $n$. Furthermore, since it is capable of only analysing every point in time individually, there is the assumption that all the shares are processed in parallel for the result of the test to match the behavior of the algorithm. This is usually the case for hardware masking implementations, that’s why it is recommended [SM15, p.12] that for software masking implementations to also perform multivariate t test to take into account the combination of different points in time to calculate the t value. The different points can be selected as the corresponding cycles of when the different shares are processed and it is possible to exactly find these points in time since the developer has white-box visibility of the design. Going back to the first-order case, the order of the univariate t test can be raised to a second order by extracting the mean-free squared traces $Y = (X - \mu)^2$ where X represent the originally acquired
measurements, \( \mu \) their mean and \( Y \) the new set of traces, so equation 4 can be applied on those new values. The same procedure can be extended to higher numbers, thus the HO-SCA can help determine the resistant against higher-order leakage.

As briefly described in this section, the assessment methodology depends on how and when the data is collected, also the data can be combined in different ways to yield other levels of observation. A special consideration must be taken when processing the shares, the accumulation of thousands of traces is required to get acceptable results, thus the computing resources like processor and memory can be directly impacted by how the test is carried out. Specially for software based masking schemes, the execution times grows with the order of the scheme and so the length of the traces grows considerably to occupy large storage. Finally, remark that the \( t \) test is not a penetration test but a detection tool that allows the designer to easily and practically find out the level of vulnerability or resistance against multiple levels of attacks.
3 Masking Implementation

This section describes the general characteristics of the masked implementation of the AES encryption algorithm and points out the considerations necessary for its proper design and resistance against HO-SCA.

The AES encryption implementation presented in this work is developed using polynomial masking and secure multiparty computations. As it was previously mentioned in section 2.2, the AES-128 encryption algorithm consists of 10 rounds of operations on a mixture of plaintext and key material known as the AES state. The linearity of the MixColumns, ShiftRows and AddRoundKey allow a straightforward translation to a masked version, however, the inherent non-linearity of the SubBytes function represents a more complicated challenge in terms of implementation, execution time and risk of leakage. The linear operations are simply applied on all the shares of the secret so the performance linearly decreases as the number of participants increases. On the other side, the SubBytes function requires square and multiplications operations that raise the order of the polynomials employed for sharing the intermediate state. To assure the resistance of the masking scheme it is fundamental to count on a true random number generator (RNG), the sharing of the secrets and the operations of the shares, like the secure multiplication, depend on the entropy of a reliable RNG.

3.1 Specifications

This work presents an 8-bit oriented implementation written in bare C. The program supports masking schemes for different $n, d$ selections to measure the performance at multiple levels. Also, there are many possibilities to make performance improvements with time-memory trade-offs especially in the GF($2^8$) arithmetic operations. The code was developed on an ARM based microcontroller ($\mu$C) although it can be easily ported to other platforms with slight changes. The platform used in this work was selected for development simplicity and flexibility although there are important challenges related to hardware constraints that are discussed in the following section.

The $\mu$C is the STM32L053R8T6 by ST-Microelectronics, it features the ultra-low power ARM Cortex-M0+ core that is a 32-bit processor capable of running at up to 32 MHz. The $\mu$C is equipped with multiple peripherals that simplify the connectivity of the development board, it supports serial communication over USART (Universal Synchronous/Asynchronous Receiver/Transmitter) port and can be connected to the USB port of a computer to establish virtual serial communication. The development board comes with the ST-Link, another micro-controller, capable of flashing and debugging the STM32 $\mu$C as well as handling USB serial communication. The STM32L053R8T6 is integrated with a hardware true RNG capable of generating a new 32-bit random number every 40 clock cycles, it has to be specifically clocked at 48 MHz to enable the generation so it requires internal phase-locked loops (PLLs) to derive a 48 MHz from the operating core clock frequency. For this implementation the selected core clock frequency is 4 MHz which comes from a high-speed external oscillator thus it has to be multiplied by 24 and divided by 2 in order to match the RNG clock frequency.

By nature, cryptographic algorithms heavily depend on random numbers to produce acceptable
results, otherwise the protected data could be easily compromised by a meticulous adversary. Furthermore, masking schemes especially rely on entropy to fuss the sensitive data processed by cryptographic algorithm implementations. For example, if the generator is pseudo-random then the adversary would be able to find out, after gathering enough information, the secret coefficients of the polynomials that are used to break the sensitive variables into shares.

The RNG has been validated following the German AIS-31 standard [STM, p.452]. This AES implementation calls either \texttt{getrn()} or \texttt{getnonzerorn()} functions everytime a new random number is required. The first one returns the least significant byte (LSB) out of the double word RNG data register regardless of its value. The second function, shown in code 1, scans the four bytes from the data register and retrieves the first non-zero byte starting from the LSB to the most significant byte (MSB). Only when the four bytes of the random number are zero, the returned byte is zero. The two functions run on constant time, but a new random number can only be generated after 40 clock cycles of a 48 MHz clock and since the core runs at 4 MHz, there's enough time to get fresh random numbers even when the functions are called back-to-back due to the function call overhead.

The reason to look for a non-zero byte in the 4-byte random number is to avoid the reduction of the degree of the sharing polynomial while generating coefficient associated with the highest degree term. If it happens to be an \((n,1)\)-masking scheme, then the shares would be the same as the secret values turning the scheme into useless execution overhead. Thus by calling the second function, the probability of getting a zero is expected to be one over four billion.

Since AES encryption algorithm is built on finite field \(\text{GF}(2^8)\) arithmetic, it simplifies software implementations as every element of the finite field can be represented with a single byte thus the result of an operation lies within the same set. Additions and subtractions on this field are trivially reduced to the XOR operation, field multiplication is a little bit more complex but there are many different ways in which it can be coded, it is a matter of finding the appropriate balance between execution time and memory usage available in the embedded device running the masked

```c
uint8_t getnonzerorn()
{
    uint8_t out = 0;
    uint8_t nonzero = 0, i, outSet = 0;
    uint8_t bytes[4];
    uint32_t rn;

    rn = RNG->DR;  // Read data register from RNG
    bytes[0] = (uint8_t)(rn >> 8);
    bytes[1] = (uint8_t)(rn >> 16);
    bytes[2] = (uint8_t)(rn >> 24);
    for (i=0; i<4; i++)
    {
        out = (˜(˜(˜outSet) & (˜(˜nonzero)) & bytes[i]) + out;
        nonzero |= nonzero >> 1;
        nonzero |= nonzero >> 2;
        nonzero |= nonzero >> 4;
        outSet |= nonzero;
    }
    return out;
}
```

Code 1: Get Non-zero Random Byte Function
encryption. In this research three different methods are used and compared in performance, in section 4.3 shows the leakage assessment of two of them. Yet, it is also possible to implement the field multiplication by applying $ab = 2^4a_h b + a_l b$ where $a$ and $b$ are the operands, and $a_h, a_l$ represent the two nibbles of $a$. Notice that $a_h b$ and $a_l b$ can be precomputed to generate two 4 kB LUTs but it is impractical from the perspective of the embedded device memory constraints, however the code for this and the rest of the functions is available on GitHub. Also, squaring in the GF$(2^8)$ field can be implemented multiple ways, a couple of them are described here. Notice that all of the functions execute in constant time, there are no branches to control the execution flow; the for loops run from beginning to end without interruption or change in the flow. Also, remark that the irreducible polynomial used in all cases is $0x1b$.

The first instance of the field multiplication, described in Code 2, consists of instructions only i.e. it has the minimum memory consumption of the three implementations but it is the slowest as well. Notice that a logic mask is used to control process of the operation, it becomes either all zeros or all ones depending on the bit that it is checking and later that mask determines whether or not to apply other logical instructions. This mechanism prevents the usage of other conditional statements that could leak information about the operands.

The second instance of the field multiplication (Code 3) saves some execution cycles by the usage of a 256-byte Look-Up Table (LUT), it turns lines 11,12 and 13 of the instructions-only multiplication in Code 2 into a single memory access. It is possible to reduce those lines because they only depend on the operand $v$, in other words, the LUT was generated using those three lines of code for all possible values of $v$. Notice that it is assumed that the table look-up is done in constant time regardless of the memory index and so it happens for the Cortex-M0+ core. There are other memory architectures where the accesses vary depending on the data location. For example, if there is a cache memory to store recently used data, then it would cause response time variations. An adversary could measure the access latency and eventually reconstruct the shared secret. However, this implementation relies on the fact that memory transactions complete in constant time and they point to addresses based on random shares. Theoretically, there should
be a power variation relative to the memory address being accessed but this SCA analysis does not reveal evidence of it.

Finally, by just using two 256-byte LUTs the Exp-Log field multiplication [GR16, p.p.5,6] offers a good time-memory trade-off. This method is synthesized by $ab = g^{\log_g(a)} + g^{\log_g(b)}$ thus by precomputing the tables $\log_g(x)$ and $g^x$ for all possible $x$, the GF(2^8) multiplication can be quickly computed. The two tables contain the same values but arranged in a different way since the two operations are opposites. However, extra code is needed to deal with the case when either of the two is zero, also the result of the arithmetic addition has to be reduced because the result could be up to $2^8 - 2$. Notice that with logical and arithmetic instructions prevent the usage of conditional statements.

A field squaring function can help boost the performance of the encryption algorithm, it is a special case of the multiplication when the two operands are the same. The easiest and fastest way of implementing it is to precompute a LUT for all the 256 possible input values, thus everytime
a squaring is done it can be easily replaced with a table look-up. It is also possible to use only instructions to compute the square of a number, Code 5 shows the necessary operations.

These elementary operations represent the building blocks to construct the masking scheme to protect AES. However, another intermediate layer of operations is necessary to implement the non-linear AES functions: multiplication and squaring of shares. They are commonly known in the literature as secure multiplication/squaring or SMC multiplication/squaring. These operations work among the shared secrets thus are more complicated and demand more execution time. Section 2.3.1, briefly introduces the sharing mechanism to break a sensitive variable into multiple parts. Section 2.3.2, describes the high-level algorithm and the required field operations to perform a secure multiplication. Notice that the performance of the secure multiplication relies on the number of shares and the degree of the polynomial selected to mask the secret variables, whereas in the secure squaring it only depends on the number of shares due to the property described in section 2.3.2.

The secure multiplication accepts two sets of shares $F$ and $G$ as input parameters, and outputs another set with the result. The product of the two secrets, from which $F$ and $G$ were derived, can be revealed by multiplying the inverse Vandermonde matrix with the resulting vector $H$. In Code 6, $N$ and $D$ correspond to the number of players and the order of the masking scheme $(n, d)$. Notice that there are conditional compilation commands based on $D$ and $\epsilon = n - 2d - 1$ (EPS) to save performance and memory for specific cases. The $\text{gfadd()}$ operation is an addition in the GF($2^8$) field, i.e. an XOR operation, that in C can be coded as a macro. Take into consideration that $\alpha_i$ for $i = 0, \ldots, n - 1$ and the inverse Vandermonde matrix are pre-computed and they are constants for all the execution of the encryption, see the appendix in section 6.

Once the properties described in section 2.3.2 are satisfied, the secure squaring functions are turned into a field squaring operations and reordering of shares. This implies a significant performance boost in terms of code size and execution time. Consider the example Code 7 specific to the $(5, 2)$-sharing scheme to illustrate the simplicity of the operation where $F[N]$ are the input shares and $H[N]$ the squared shared secret.

Masking schemes represent a big performance workload to embedded devices and there are many mathematical approaches, from the threshold schemes theory and architectural shortcuts perspective, that can be used to speed up their execution. The C implementation does not allow to fully leverage the intrinsic micro-architectural benefits of the ARM architecture, most of it is left to the compiler. On the other hand, the implementation can be easily ported to different

```c
uint8_t gfsqr(uint8_t a)
{
    return (a & 0x01) ^
            (a & 0x02) << 1 ^
            (a & 0x04) << 2 ^
            (a & 0x08) << 3 ^
            (a & 0x10) >> 4 & 0x1b ^
            (a & 0x20) >> 5 & 0x6c ^
            (a & 0x40) >> 6 & 0xab ^
            (a & 0x80) >> 7 & 0x9a;
}
```

Code 5: GF($2^8$) Squaring. Lines 2, 3, 4 and 5 square every bit of the lower nibble. Lines 6, 7, 8 and 9 square every bit of the upper nibble and reduce the partial result.
void Multiplication(uint8_t F[N], uint8_t G[N], uint8_t H[N])
{
    uint8_t i, j, a, sum, sum1;
    uint8_t tmp1, tmp2;
    uint8_t Q[D+1];
    uint8_t QFunctions[N][N];

    #if D>1
        uint8_t t;
    #endif

    for (i=0; i<N; i++) {
        Q[0] = gfmult(F[i], G[i]);
        Q[j] = getrn();
        for (j=1; j<(D+1); j++) {
            Q[j] = getrn();
            sum1 = Q[0];
            #if D==1
                sum1 = gfadd(sum1, gfmult(Q[1], alpha[a]));
            #else
                for (t=1; t<(D+1); t++) {
                    sum1 = gfadd(sum1, gfmult(Q[t], gfexp(alpha[a], t)));
                }
            #endif
            QFunctions[i][a] = sum1;
        }
    }

    for (j=0; j<N; j++) {
        sum = 0;
        for (i=0; i<N; i++) {
            tmp1 = gfmult(InvVand[0][i], QFunctions[i][j]);
            if (j<D) {
                tmp2 = gfmult(InvVand[N-j-1][i], gfmult(F[i], G[i]));
                sum = gfadd(sum, gfadd(tmp1, tmp2));
            } else {
                sum = gfadd(sum, tmp1);
            }
        }
        H[j] = sum;
    }

    Code 6: Secure Multiplication.

void qPow2(uint8_t F[N], uint8_t H[N])
{
    H[0] = gfsqr(F[2]);
    H[1] = gfsqr(F[3]);
    H[2] = gfsqr(F[1]);
    H[3] = gfsqr(F[0]);
    H[4] = gfsqr(F[4]);
}

Code 7: Secure Squaring in the (5,2)-sharing scheme.
platforms and processor architectures due to the omnipresence of embedded C in the industry. The following section focuses on the performance of the ARM Cortex-M0+ core running the masking scheme.

### 3.2 Performance

The performance measurements are described for two selections of $n$ and $d$ that satisfy $n = 2d + 1$. For practical purposes the (3,1)-sharing scheme is the lowest that meets this condition and with the 4 MHz frequency of the core it represents a realistic case to measure performance. Since the order of the scheme is one, it can be broken with a second order side-channel analysis but still it is worth understanding the practical implications on an embedded device. The (5,2)-sharing scheme is also analyzed, it represents a more realistic case that could be potentially employed for embedded applications. The way the code is written, it is possible to select other modes like (4,1), (5,1) or (6,2) however these selections would not bring higher resistance to SCA compared to the (3,1) and (5,2).

The execution time of the two schemes is shown in Table 1, the data is given in $\mu$s and the system clock employed for the measurements was 4 MHz. Only AES-128 encryption has been implemented, however AES-192, AES-256 encryption and decryption should be a straightforward process. Remark that different versions of the GF($2^8$) multiplication and squaring were used, even though only a few permutations were selected to show the performance comparison, the user can select any combination of the field operations to meet the embedded device’s limitations. As a reference, consider the OpenSSL 1.0.1g AES encryption released on April 2014. It is a 32-bit C implementation compiled for the ARM Cortex-M0+ to run at 4 MHz. The execution time for this unmasked encryption is 481.5 $\mu$s, Table 5 shows its corresponding code and data size. Notice that, even though full unrolling is disabled, the code and data size is significantly large, in return the execution time is 1090X faster than the shortest masked encryption in Table 1.

The breakdown of SMC operations per round of AES is shown in Table 2, it describes how many SMC operations are needed for every function of the AES encryption algorithm.

The SubByte function requires SMC operations, i.e. secure multiplications, secure squaring, additions and affine transformations due to its non-linear nature. In turn, these SMC operations are built on top of GF($2^8$) operations. Table 3 describes the necessary field operations for every SMC operation.

Finally, Table 4 shows the execution time required for the field operations and the SMC multi-

---

### Table 1: Execution time for AES-128 encryption in $\mu$s with the CPU running at 4 MHz.

<table>
<thead>
<tr>
<th></th>
<th>(3,1)</th>
<th>(5,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr. Only</td>
<td>1.453.107,0 1.119.818,0 524.896,0</td>
<td>8.047.896,00 6.487.622,00 2.903.750,0</td>
</tr>
<tr>
<td>LUT</td>
<td>1.597.706,0 1.239.214,5 579.480,0</td>
<td>8.621.960,50 6.977.620,00 2.956.370,0</td>
</tr>
</tbody>
</table>

1 Encryption based on error preserving multiplication which requires more operations, more details in [SES17].
Table 2: Number of SMCs in one round of AES where SubBytes is split into two parts Inv(y) inversion and τ(y) affine transformation over GF(2^8).

<table>
<thead>
<tr>
<th>SMC Multiplication</th>
<th>Inv(y)</th>
<th>τ(y)</th>
<th>MixColumns</th>
<th>AddRoundKey</th>
<th>ShiftRows</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 × 4</td>
<td>16 × 3</td>
<td>16 × 7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Efficient Squaring</td>
<td>-</td>
<td>16 × 7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SMC Addition</td>
<td>-</td>
<td>16 × 9</td>
<td>12</td>
<td>16 × 1</td>
<td>-</td>
</tr>
<tr>
<td>Affine Transformation</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3: GF(2^8) field operations required for SMC operations where ϵ = n − 2d − 1.

<table>
<thead>
<tr>
<th>SMC operation</th>
<th>Multiplication</th>
<th>Squaring</th>
<th>Addition</th>
<th>Affine Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Mul.</td>
<td>n^2(d + 1) + n(ϵ + d + 1)</td>
<td>n</td>
<td>-</td>
<td>n</td>
</tr>
<tr>
<td>Field Add</td>
<td>n^2(d + 1) + n(ϵ + d)</td>
<td>-</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Randomness</td>
<td>nd</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

plication, it also includes the random number generation functions. Notice that based on Table 3, these building block operations represent the key elements to boost the performance of the masking scheme, that is the reason to look for faster methods to do field arithmetic.

The performance results show that SMC multiplication is mainly the bottleneck of the encryption algorithm and in turn it relies on the field multiplication. The execution time can be also reduced by running at higher frequencies, this board is capable of running at 32 MHz, however power consumption increases with the CPU frequency. Table 5 shows the different sizes for code and RW-data according to selected combinations of field operations, notice that other combinations are also possible to produce different code and data sizes.

3.3 Challenges

The Nucleo-L053R8 board has features that allow the rapid implementation of the masking scheme. The ARM architecture omnipresence and the development tools are valuable advantages to bring the concept to life, however important considerations must be taken especially when designing performance hungry algorithms into a resource constrained device.

The µController has an embedded RNG that it’s fundamental for the secure operation of the scheme. The downside of it is that it has to be clocked at the particular frequency of 48 MHz.

Table 4: Execution time for GF(2^8) and SMC operations in μs with the CPU running at 4 MHz.

<table>
<thead>
<tr>
<th></th>
<th>(3,1)</th>
<th>(5,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GF(2^8) mult.</td>
<td>54,50</td>
<td>44,50</td>
</tr>
<tr>
<td>GF(2^8) sqr.</td>
<td>13,75</td>
<td>1,50</td>
</tr>
<tr>
<td>getrnd()</td>
<td>3,75</td>
<td>3,75</td>
</tr>
<tr>
<td>getnonzerorn()</td>
<td>41,75</td>
<td>41,75</td>
</tr>
<tr>
<td>SMC mult.</td>
<td>1.247,75</td>
<td>1.026,25</td>
</tr>
<tr>
<td>SMC mult!</td>
<td>1.427,50</td>
<td>1.175,50</td>
</tr>
</tbody>
</table>

1 Error preserving multiplication which requires more operations, more details in [SES17].
Table 5: AES-128 encryption code and RW-data size depending on the GF(2^8) operations variations.

<table>
<thead>
<tr>
<th></th>
<th>unmasked</th>
<th>(3,1)</th>
<th></th>
<th>(5,2)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GF(2^8) sqr.</td>
<td>-</td>
<td>Instr. Only</td>
<td>LUT</td>
<td>LUT</td>
<td>Instr. Only</td>
</tr>
<tr>
<td>Code Size</td>
<td>7.23 kB</td>
<td>3.38 kB</td>
<td>3.26 kB</td>
<td>3.29 kB</td>
<td>3.58 kB</td>
</tr>
<tr>
<td>RW-data</td>
<td>12 B</td>
<td>12 B</td>
<td>524 B</td>
<td>780 B</td>
<td>32 B</td>
</tr>
<tr>
<td>RO-data</td>
<td>8.7 kB</td>
<td>224 B</td>
<td>224 B</td>
<td>224 B</td>
<td>224 B</td>
</tr>
</tbody>
</table>

and the user can configure the clock tree to get the dedicated High-Speed Internal oscillator for it, however it is not a trivial task.

It is also possible to provide the system clock from various sources, either internal or external but there are important considerations regarding side-channel analysis that are going to be described in section 4.2. For this implementation, the clock generation comes from an external crystal and it derives to a PLL module that generates the desired 48 MHz for the RNG. The external oscillator can be chosen for different frequency values because the PLL can multiply and divide that frequency to generate another one.

Since serial communication is fundamental for the development and analysis of the scheme, some libraries are reused and other functions were created for this work. The board comes with the ST-Link chip that allows USB communication between the target µC and the PC, it is used to flash and debug the µC. According to the clock source and its frequency, the USART module and the internal clock tree have to be configured appropriately to establish serial communication.

There are a few tool chains available to develop the code and debug it. This work was initially implemented on mbed that is an online compiler provided by ARM available to all registered users, registration is free of charge and there’s no limitation on the amount of code that can be compiled. It also has an integrated version control tool to keep track of the progress and allow other users to collaborate in the development life cycle. This tool chain allows the users a rapid development flow since most of the basic configurations are given by default and there are many libraries available. Unfortunately it does not offer a debugger and internet connection is needed to use the compiler. On the other end, the KEIL development kit is more complete, it gives the user more control over the configuration of the device, it allows inline assembly and includes a debugger. The software can be downloaded for free, however registration is required and it allows to flash up to 32 kB, which can be a restriction if large LUTs are used alongside with the code.

From the implementation perspective these are the major challenges, later in the next section other challenges that rose during the leakage assessment process are described.
4 Leakage Assessment

Theoretically, an \((n,d)\)-sharing masking scheme is resistant to a \(d\)-order side-channel analysis. This work focuses on two masking levels (3,1) and (5,2), for both of them, it shows the \(t\) test results of the power traces taken during the secure multiplication. This operation is one of the key components in the non-linear function of AES, it is also the most complex operation throughout the whole encryption thus it represents an attractive entry point for an adversary. Even though there are multiple techniques, like Simple Power Analysis (SPA) and Differential Power Analysis (DPA), than can help an adversary to break the cryptographic implementation, the \(t\) test reveals any possible source of leakage and the point in time where it is happening.

4.1 Setup

The setup consists of the design under test (DUT) which is the NUCLEO-L053R8 development board by ST-Microelectronics, a PC to interact with the DUT via serial communication, a LeCroy WavePro 725Zi oscilloscope, an active differential probe LeCory AP 033, a regular probe LeCory PP007-WR and an external power source. The differential probe is connected to the JP6 pins of the board and a small 47Ω resistor is connected between the two pins to sense a small voltage drop but still leave enough voltage for the \(\mu\)C to work. The regular probe is used to synchronize the recording of the trace with the beginning of every multiplication, one of the GPIOs is used for this purpose tu output a signal that is set before the operation and reset after it is finished. The oscilloscope starts recording when the trigger signal is detected, it stops after a defined period of time enough to capture the processing of the whole operation. The traces are saved in a hard disk drive and then processed in Matlab sequentially.
4.2 Challenges

The computation of the t test might seem trivial but in reality there are some considerations that need to be taken properly to get accurate results in an efficient way. The nature of side-channel analysis requires a large amount of samples to properly identify possible sources of leakage, in other words, if the size of the traces is large, storing and processing them can represent a big effort. Here are the main challenges encountered while doing the leakage assessment of this masking scheme implementation.

4.2.1 Trace Generation

As mentioned in section 2.4 two different sets of traces have to be generated to avoid false positive results, the approach is fixed-vs-random. First, a script running on the PC generates a random number and based on its value either a fixed secret value or a random secret value is broken into shares by polynomial masking and then the corresponding shares are fed into the secure multiplication function. It is essential to have continuous communication between the DUT and the PC to send the random execution pattern, also the PC must keep track of all the patterns and the order in which they were generated to process the traces appropriately.

To eliminate all possible sources of interference, the unnecessary modules of the microcontroller are turned off. Even the USART that handles the serial communication is switched off during the secure operation and restored right after the multiplication completes. Another prevention is delaying the processing of the secure operation for a few micro-seconds after the trigger that synchronizes the storage.

An important requirement for the implementation is to have a constant execution time otherwise it could leak information about the internal state. From the programming perspective, this implementation runs in constant time however there’s an important factor that is taken for granted: the clock cycles are always constant. Execution time does not only depend on the instructions and

Figure 7: Lab. Setup: NUCLEO-L053R8 board (center), oscilloscope (upper left), external power source (left), PC (right) and external hard disk (upper right).
the flow control mechanisms but on the clock precision as well. One of the biggest issues found during the early stages of the analysis was that the traces were misaligned, the misalignment was caused by clock jitter. In other words, the oscillator that the µC was using as clock source was an internal RC resonator thus the precision of the cycles was very low and it produced different trace lengths. Fixing that problem is possible but it would have required advanced techniques like Elastic Alignment [vWWB11]. A simpler solution was to configure the internal clock tree of the µC to use an external high-speed oscillator as a clock. A 4 MHz crystal was chosen to test the (3,1)-sharing scheme and a 16 MHz crystal for the (5,2) case. After that workaround all the traces were correctly aligned and ready for the analysis.

4.2.2 Trace Collection

Masking schemes involve a redundant amount of work thus performance is a very important factor, especially in single-threaded software implementations where the workload is hardly parallelized. Another downside is the relatively slow frequency at which embedded devices run compared to FPGAs, ASICS or high-end computers. Thus the execution time of such implementation is large for low-power embedded devices, the length depends on the degree of the masking scheme, the number of shares employed, the frequency of the device’s clock and the performance of the algorithm. The longer the execution time, the larger the traces are going to be. However, it also depends on the settings of the oscilloscope like the sampling rate and the way each point is represented in a trace file. For example, the size of a trace file for a multiplication that lasts around 1 ms is approximately 100 KB, if the analysis requires hundreds of thousands of traces or millions, then storage becomes a significant parameter when dealing with higher-order masking schemes and analysis.

4.2.3 Trace Processing

Due to the huge amount of data needed for the analysis and the length of each trace, an efficient way of computing the t test is necessary. The test itself requires the average and variance of two large sets, it is infeasible to accumulate all the traces in DRAM memory to do those calculations, eventually the PC would run out of memory and the Matlab script would crash. Thus it is important to compute the t test progressively. Instead of calculating the mean and variance of the whole sets, one can process a certain amount of traces at a time, without overflowing the memory, and compute the local mean, the local variance and accumulate them for all the subsets. If these subsets are large enough, the accumulated mean and the variance would tend to be equal to the total mean and variance of the whole sets and the t test would bring valid results.

It is also possible to delimit each of the traces to length of the effective trace. Oscilloscopes usually have fixed ranges of trace recording that cause larger traces than necessary and if t test does not set a starting point and a limit on each of the traces then part of the computation will be wasted. Matlab also offers parallel processing to improve the performance of the test, however it is not always possible or necessary to reach that level but it is recommended to use vector and matrix operations as much as possible since they have been designed with performance in mind.

Finally, The bandwidth of the oscilloscope can be determining factor when computing the t test. If the bandwidth is very high, the trace will contain information regarding high frequencies beyond
the operating frequency of the embedded device and thus yield misleading results. For the results of the experiments presented in this work the bandwidth of the differential probe was set to 20 MHz, also the samples are taken at a 100 million samples per second without any attenuation or gain factor in the probe.

4.3 SCA Results

This section contains the results on the side-channel analyses performed on the (3,1)- and (5,2)-sharing masking schemes, only the secure multiplication has been analyzed due to its relevance in the non-linear function of AES encryption. The traces were taken under the same test conditions, the only difference between the two cases is the frequency of operation of the target device.

4.3.1 Higher-Order t test

The leakage assessment of the secure multiplication for the (3,1)-sharing is shown in Figure 8, the system clock ran at 4 MHz, it used the field multiplication with minimal memory footprint thus the execution time is the longest of the provided versions. The plot shows the orders from the first to fifth, notice that for this version of the field multiplication, there’s a significant leakage at various points under the second and fourth order analysis. For all the plots, the red upper and lower lines represent the absolute 4.5 bounds, if the result lies within that range, it can be considered secure based on the number of traces.
The Exp-Log field multiplication offers a 2.6X execution speedup over the whole secure multiplication, although it implies the allocation of two 256-byte LUT, the leakage assessment in Figure 10 is contained within the acceptable boundaries. As for the results, it seems to be better suitable
for side-channel resistance compared to the previous Figure 8.

![Graph](image.png)

Figure 10: HO t test for SMC (3,1)-multiplication with Exp-Log GF($2^8$) multiplication.

Figure 9 shows the t growth over the number of samples for the first order t test, Figure 11 shows the corresponding values for the (3,1) secure multiplication based on the Exp-Log field multiplication.
Finally, Figure 12 shows the $t$ result for the (5,2)-sharing scheme. Due to its execution length with a 4 MHz clock that would turn the trace collection impractical, the system clock was switched to 16 MHz. This SMC multiplication uses the Exp-Log field multiplication as well.
Based on this analysis it can be inferred that the univariate first order t test does not show points of leakage, the higher order tests reveal some points where there could be certain leakage. It is important to mention that although the amount of traces is relatively small, for many experiments previous to these results the test revealed significant leakage at different points. Thus, based on experience and considering the internal sources of noise in the \( \mu C \), these results show a good level of resistance. However, single-threaded software masking schemes process one share at a time, thus it is important to test it under a multivariate analysis to check if there’s any leakage due to the relation of two different points in time.

4.3.2 Multivariate \( t \) test

As opposed to hardware implementations of secret sharing and multiparty computation that process their shares in parallel, this is a single-threaded software implementation. The operations on every share or pair of shares is done sequentially, so it happens that the power consumption at certain interval may only be related to a single share or pair of shares being processed [SM15, p.12]. This section describes how the multivariate \( t \) test was applied to the collection of traces previously used for the univariate \( t \) test in section 4.3.1 and the results for the previously introduced cases.

Once the traces are classified in two sets (fixed vs random), their corresponding mean for every point in time is calculated and substracted from every sample. That generates two new meanless sets of traces containing the same amount of samples as the previous sets. Later, to save execution time and memory consumption, a few intervals of time are identified where the different shares are processed. Then all the points are multiplied with each other to generate all possible combinations and proceed to calculate the \( t \) test with these new data. Notice that doing the multivariate analysis on a section against itself would yield the second order univariate \( t \) test as part of the result.

Figure 14 illustrates this task by pointing out relevant events throughout the execution of a (3,1) secure multiplication. For example, the rising and falling edge of the blue signal marks the beginning and end of the first three field multiplications within the SMC multiplication. During these three sections, the shares of the operands are processed for the first time. In other words, each section corresponds to a field multiplication according to line 12 of Code 6, thus these are
areas of interest to focus the multivariate analysis on. This analysis demands a great amount of memory because the objective is to combine (multiply) one point with all the others, so the size of data grows from \( n \) to \( n^2 \) where \( n \) is the number of sample points under analysis.

Figure 14: The black trace is a signal that is set before the beginning of the SMC multiplication and reset by the end of it, the blue trace is a signal that is set by the beginning and reset by the end of each of the first three field multiplications. The gray signal in the background is the first order \( t \) result for the overall \((3,1)\) SMC multiplication based on instructions-only field multiplication.

Figures 15 and 16 show the results of the multivariate analysis on relevant sections of the \((3,1)\) secure multiplication. Each of the plots belongs to the combination of the section where the first pair of shares is processed to the sections where the remaining pairs are processed as line 12 of Code 6 describes.
Figure 15: Multivariate t test for sections of the (3,1) SMC multiplication based on Instructions-Only GF($2^8$) multiplication. Share-1 vs Share-1 shows the multivariate t test result of all combinations of points during the first GF($2^8$) multiplication. Share-1 vs Share-2 shows the result of the multivariate t test for the combination of points from the first field multiplication to the second one. Share-1 vs Share-3 corresponds to the multivariate t test analog to the previous case. Share-1 vs Other shows the result of multivariate t test of the points during the first field multiplication combined with all the points of a section close to the end of the SMC multiplication.

Notice that the graphs in Figure 15 reveal certain peaks indicating potential leakage, these do not appear under the univariate high-order analysis. It could be possible that the same internal hardware, like registers, is exercised at those two points in time and there is a relevant difference in the average power consumption.
Figure 16: Multivariate t test for sections of the (3,1) SMC multiplication based on Exp-Log GF(2^8) multiplication. Share-1 vs Share-1 shows the multivariate t test result of all combinations of points during the first GF(2^8) multiplication. Share-1 vs Share-2 shows the result of the multivariate t test for the combination of points from the first field multiplication to the second one. Share-1 vs Share-3 corresponds to the multivariate t test analog to the previous case. Share-1 vs Other shows the result of multivariate t test of the points during the first field multiplication combined with all the points of a section close to the end of the SMC multiplication.

Despite the fact that the Exp-Log field multiplication uses table look-ups, the result in Figure 16 does not show any evidence of leakage derived from the memory accesses. In fact, the results are more secure than those of Figure 15. Also, the analysis could be extended to other areas of interest in the secure multiplication that may reveal potential leakage.

Figure 17 shows the corresponding analysis output as in previous figures but for the (5,2) secure multiplication. In this case, notice that there are possible leakage points, one of the reasons is that the frequency of the clock is 16 MHz, instead of the 4 MHz in the previous cases. A higher frequency reduces the time interval between one state and the next in the hardware thus causing the power consumption to overlap. As this is an on going research, the detected leakage is taken into consideration for future analysis. Once those points of interest have been discovered, a more detailed analysis can be made to gather more information.

The multivariate analysis is a useful tool to reveal potential sources of side-channel leakage. However the time execution and memory constraints are significant factors to constrain the exten-
sion of the analysis to certain sections. Remark that for all of the multivariate analysis results, the horizontal axis does not represent time since the analysis itself requires the combination of traces at different points. The data can be plotted in a 3D format to make it visually easier to understand and identify the exact points of leakage, however for the simplicity of this document the data is expanded across the horizontal axis.

![Figure 17: Multivariate t test for sections of the (5,2) SMC multiplication. Share-1 vs Share-1 shows the multivariate t test result of all combinations of points during the first GF(2^5) multiplication. Share-1 vs Share-2 shows the result of the multivariate t test for the combination of points from the first field multiplication to the second one. Share-1 vs Share-3, Share-1 vs Share-4, Share-1 vs Share-5 correspond to the multivariate t test analog to the previous case. Share-1 vs Other shows the result of multivariate t test of the points during the first field multiplication combined with all the points of a section close to the end of the SMC multiplication.](image-url)
5 Conclusion

The omnipresence of embedded devices in every sector of our lives opens many gaps in terms of security. Even standard algorithms for encryption, authentication and integrity checking are susceptible to side-channel analysis, not because of any possible intrinsic weakness in its design but due to their utilization on real hardware that scapes the safety of their conceptualization. SCA resistance comes at a performance and development cost, masking schemes pose complex challenges on the hardware architecture in order to securely implement these algorithms.

Compared to high-end computers that have adopted AES instructions part of their Instruction Set Architecture (ISA) extensions that can execute a whole AES encryption in just a few clock cycles, or hardware modules that are integrated into a huge variety of embedded devices that parallelize the encryption workload, polynomial masking and SMC fall short in terms of execution time. However, as described in this work and others e.g. [GR16], it is possible to optimize masking schemes from both theoretical and implementation perspective. SMC and field operations can be implemented in multiple ways to speedup execution but it always boils down to the hardware architecture and the desired level of resistance.

Yet there are many considerations that have to be correctly handled to achieve proper impermeability. From the essential branchless design that already demands a relatively small overhead, and the reliance on true randomness generation to disperse the secrets appropriately, the developer must take care of fitting the secure algorithm into the constrained hardware and still allow space for the application itself. Hardware features can help boost the performance, for example, instruction set operations like cmov that does conditional movement of data in constant time or the presence of an embedded True Random Number Generator simplify the implementation.

Software based masking schemes are significantly slower than their hardware counterparts, that implies a higher effort in terms of side-channel trace generation, collection and processing. The clock sources and the available frequencies of operation of the embedded device influence the complexity of SCA. Dozens of gigabytes of storage are necessary to save the power traces and their collection takes hours and in some cases even days. A slight error in the setup, like a serial communication error, can turn the trace collection into trash. However, once the observer has become familiar with the implications of the analysis, some improvements can be made to it. Like reducing the trace recording to the effective processing time or by the automation of repetitive work with scripts.

This thesis gives evidence of the feasibility and side-channel resistance of polynomial masking and SMC in software, they demand a performance overhead but in return provide an acceptable level of impermeability. The analysis can be extended but it already comprehends key elements, like the focus on secure multiplication, randomness and constant time execution with small but practical orders of protection. The code has been made publicly available, it counts with several selectable optimizations that imply a trade-off in memory and processing time. This contribution may not be considered a ground breaking work but it can help as a reference to further extend the side-channel resistance to software implementations.
6 Appendix

```cpp
#el {if (N==5) && (D==2)
    uint8_t alpha[N] = {0x51, 0xec, 0x0d, 0xb1, 0x01};
    uint8_t InvVand[N][N] = {
        {0x01, 0x01, 0x01, 0x01, 0x01},
        {0x5d, 0x5c, 0xe0, 0xe1, 0x00},
        {0xbc, 0xbc, 0xbd, 0xbd, 0x00},
        {0xb1, 0x0d, 0x51, 0xec, 0x01},
        {0x51, 0xec, 0x0d, 0xb1, 0x01}
    };
#}n if (N==3) && (D==1)
    uint8_t alpha[N] = {0x01, 0xbc, 0xbd};
    uint8_t InvVand[N][N] = {
        {0x01, 0x01, 0x01},
        {0x01, 0xbd, 0xbc},
        {0x01, 0xbc, 0xbd}
    };
#e

Code 8: Precomputed selection of evaluation points ($\alpha_i$) and Inverse Vandermonde Matrices for (3,1)- and (5,2)-sharing.
```
References


37


