Aleksander Waage

Secure Implementation of a RISC-V AES Accelerator

Master's thesis in Electronic Systems Design
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Henrik Fegran
June 2022

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Aleksander Waage

Secure Implementation of a RISC-V AES Accelerator

Master’s thesis in Electronic Systems Design
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Henrik Fegran
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Project Description

The CV32E40-family of CPU cores are a set of open-source CPU cores originally spawned from the RI5CY project, and as with any modern computing system, security is considered paramount.

No proprietary code and no personal information is safe from prying eyes in the absence of cryptographic protection, but the cryptographic algorithms themselves are not sufficient to prevent disclosure of confidential information.

As cryptanalysists are facing difficulties in breaking the algorithms, much of the research focus has shifted from the algorithms themselves to potential weaknesses in the cryptographic implementations.

The goal of this master thesis is to design and implement a secure accelerator for AES, implementing countermeasures for common classes of exploits, for use with the CV32E40 family of Risc-V CPU cores.

This accelerator should adher to the recently ratified Zkne (AES encryption) and Zknd (AES decryption) extensions as defined by the RISC-V scalar cryptography specification.

Abstract

Technology has become a significant part of our daily lives, and it is expected to be trustworthy and secure. Security in new technologies has become a critical element in every design step for both hardware and software. High-level security techniques attempt to make information inaccessible, and lower levels try to keep information secure even if it is accessed by, for example, obfuscating it. Obfuscation is typically done using cryptographic algorithms such as the Advanced Encryption Standard (AES), which has become one of the most widely used ciphers since the endorsement from National Institute of Standards and Technology (NIST). Its operations are simple and can be performed in software and hardware, where software implementations require less specialized resources and hardware accelerators are typically faster. However, researchers uncovered that hardware could leak information through side-channels such as power consumption. It is possible to deduce critical information from these leakages through statistical analysis referred to as side-channel attacks (SCA). Several countermeasures have since been developed against such attacks, where the typical ones for AES are threshold implementations and variants of it.

In this work, a first-order Side-Channel Attack (SCA) secure Advanced Encryption Standard (AES) accelerator has been implemented. The design is based on the instructions from the official RISC-V scalar-crypto instruction set extension (ISE). A 2-share Domain-Oriented Masking (DOM) scheme is implemented to secure the accelerator against first-order SCAs. The adversary must acquire information from all shares to get any meaningful data, which is a consequence of splitting the input into multiple shares. It is simple to implement share-splitting for linear operations but not for non-linear operations such as the AES SBox. The DOM scheme was selected to make the Substitution-Box (SBox) secure against first-order SCAs. The accelerator has two parallel paths for linear operations and shares the DOM SBox for all parallels. The accelerator is connected to the RISC-V CV32E40X core by the OpenHW group through their novel eXtension Interface (XIF). A wrapper integrates the accelerator and logic required to accept incoming instructions and write back results.

The area of the complete accelerator is 2.6 times larger compared to the unprotected implementation available in the scalar-crypto Instruction Set Extension (ISE). The throughput is also decreased by a factor of 6 due to the pipeline. All the designs in this work are available on GitHub.

Sammendrag

Teknologi har blitt en viktig del av hverdagen, og det forventes at det er sikkert å bruke det. Sikkerhet er dermed en kritisk faktor i hvert steg under utviklingen. Konfidensiell informasjon blir ofte skjult ved bruk av en kryptografiskalgoritme som kan konvertere data til en uforståelig kryptert form. En dekrypteringsprosedyre kan konvertere den tilbake til sin opprinnelige form når det trengs. Avansert krypteringsstandard (AES) har blitt en av de mest brukte krypteringsalgoritmene siden den ble standardisert av Nasjonalt Institutt for Standard og Teknologi (NIST) i år 2000. AES består av flere enkle operasjoner som kan bli utført i både programvare og i maskinvare. En akselerator implementert i maskinvare er typisk raskere enn å eksekvere instruksene i programvare, men krever spesialdesignede komponenter. Det ble i senere tid oppdaget at maskinvare lekker informasjon gjennom sidekanaler som strømforbruket. Strømforbruket til en krets er sterk korrelert til hvor aktiv kretsen er, som kan føre til tydlige og observerbare karakteristikker. Sidekanalsangrep bruker statistisk analyse til å avdekke hemmlig informasjon fra disse lekkasjene. Det finnes i dag flere tiltak for å beskytte mot slike angrep, hvor de mest typiske tiltakene er terskelimplementeringer og varianter av dem.

I denne oppgaven har en AES akselerator, som er beskyttet mot første-ordens sidekanalsangrep, blitt utviklet. Akseleratordesignet er basert på den forventede oppførselen definert i RISC-V scalar-crypto instruksjonssetutvidelsen. Den tilførte verdien blir splittet i to deler, hvor den ene delen er summen av inngangsverdien og en tilfeldig verdi, og den andre delen er den tilfeldige verdien. Dette fører til at en motpart må kombinere informasjon fra begge disse signalene for å avdekke noen brukbare data. Designprosessen ble delt inn i tre deler. Først var fokuset å implementere den ikke-lineære DOM SBoksen, siden den er den største og mest kompliserte komponenten for AES. Denne SBoksen ble så brukt i den komplette akseleratoren som ble utviklet etterpå. Tilslutt ble akseleratoren pakket inn i en modul som kobler den til CV32E40X kjernen gjennom utvidelsesgrensesnittet.

Arealet på akseleratoren sammen med innpakningen er 2,6 ganger større enn eksempelimplementasjonen gitt i scalar-crypto instruksjonssetutvidelsen. Gjennomstrømningen av data er også 6 ganger saktere for en ikke optimalisert applikasjon på grunn av pipelinedesignet.

Acknowledgements

I would like to express my deepest appreciation to my supervisors Henrik Fegran (Silicon Labs) and Per Gunnar Kjeldsberg (NTNU). Our meetings were light and insightful, so I was always excited to get your input. Your thorough feedback helped me elevate this work by a lot. Special thanks to Henrik for your technical expertise that helped me when I was stuck and to consider different alternatives. And to Per Gunnar for ensuring that I was moving in the right direction and continuously developing my thesis.

Finally, I would like to thank family and friends who have given me encouraging words and good feedback throughout the semester.

Contents

List of Tables

List of Figures

List of Listings

List of abbreviations

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
CMOS	Complementary Metal-Oxide Semiconductors
CPA	Correlation Power Analysis
CPU	Central Processing Unit
DES	Data Encryption Standard
DOM	Domain-Oriented Masking
DPA	Differential Power Analysis
FPGA	Field-Programmable Gate Array
FIFO	First-in-First-Out
FIPS	Federal Information Processing Standards
GE	Gate Equivalent
GF	Galois Fields
HDL	Hardware-Descriptive Language
HO-DPA	Higher-Order Power Analysis
ID	Instruction Decode
IF	Instruction Fetch
IoT	Internet-of-Things
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
IP	Intellectual Property
MMIA	Multivariate Information Analysis
NIST	National Institute of Standards and Technology
ROM	Read-Only-Memory

RSA	Rivest-Shamir-Adleman
RNG	Random Number Generator
SBox	Substitution-Box
SCA	Side-Channel Attack
SPA	Simple Power Analysis
TI	Threshold Implementation
WAR	Write-Wfter-Read
XOR	Exclusive-Or
XIF	eXtension Interface

Chapter 1

Introduction

With technology becoming such a quintessential part of everyday life, it is critical that it is trustworthy. On one side, researchers and developers are creating better, faster, and more complex technologies to improve upon previous versions. While on the other side, the designs are thoroughly scrutinized for weaknesses that are abusable by adversaries. When developing new technologies, it is critical to consider security at every step. One of the techniques used to secure information is by obfuscating and hiding it in plain sight. A cipher can be used to encrypt information using a secret key and can be reversed through a decryption procedure to recover the original information. Ciphers are either asymmetric or symmetric; asymmetric ciphers use different secret keys for encryption and decryption, while symmetric ciphers use the same key for both procedures. Two well-known ciphers are the Rivest-Shamir-Adleman (RSA) [1] asymmetric cipher and the Advanced Encryption Standard (AES) [2] symmetric cipher. Some ciphers can be implemented in both software and hardware; software implementations use typical processor instructions, while hardware implementations require specialized units. However, implementing it in hardware typically results in a significant faster implementation [3].

Modern ciphers are typically sound against statistical attacks on their data [4]. Attempting to guess the secret key, referred to as a brute-force attack, is infeasible with the large secret key sizes. In 1998, *Paul Kocher et al.* [5] found that hardware implementations leak information through physical parameters such as power consumption, also called side-channels. This article also presented two Side-Channel Attack (SCA) that could be used to deduce critical information from this leakage. Today, simple SCAs have become ubiquitous, with resources on how to perform them and special hardware being available online [6], and the maturity of higher-order attacks is increasing. As a consequence, it is critical to be aware of state-of-the-art attack vectors and the direction the development is heading towards when developing new technologies. The level of security should be defined during the initial requirements phase to be able to get a good balance of cost and security. A straightforward RSA or AES hardware implementation can be vulnerable to side-channel leakages [5][7]. Various means have been employed as countermeasures against such attacks, where those that are mathematically proven to be sound have become the norm for modern implementations, which includes schemes such as masking [8], Threshold Implementation (TI) [9], and variations of TI [10][11][12]. In 2019 National Institute of Standards and Technology (NIST) announced a project to standardize threshold implementations for ciphers [13].

RISC-V is an open-source Instruction Set Architecture (ISA), which has increased in popularity since the establishment of the RISC-V foundation. An increasing number of companies and developers are adopting and contributing to the development, which results in a more mature ISA. The OpenHW group, whose mission is to develop free and high-quality open-source technology [14], has adopted RISC-V. One of their projects is the CVE4 family, where Silicon Labs have been a major contributor, consisting of four RISC-V cores for various embedded applications [15]. CV32E40X is one of the cores in this family that is aimed at compute-intensive applications. It implements a novel eXtension Interface (XIF) that allows custom instructions or additional Instruction Set Extension (ISE)s to be connected to the core pipeline as coprocessors.

One of the key strengths of RISC-V is its modularity. There are five official base instruction sets containing the minimum number of instructions required in a core design, which can be further extended by custom or official ISE. One newly ratified ISEs is the scalar-crypto extension that defines instructions and their corresponding behavior for various cryptographic algorithms such as AES. The AES implementation available online [16] is small and efficient but it does not implement any countermeasures against side-channel leakages and could be vulnerable to SCAs.

1.1 Objectives and Limitations

The objective of this work is to implement a secure AES accelerator. The approach for achieving this goal is split into multiple phases. The first phase consists of researching the standard AES to understand how it works and can be implemented in hardware. A high-level model was developed in python to help deepen the understanding of it and is available in [17]. In parallel, information leakage, side-channel attacks, and countermeasures were studied. A simple XIF wrapper, using the accelerator from the scalar-crypto ISE, was first designed to get started with the core-v-verification environment [18] and develop a C application using the AES instructions. After selecting the masking scheme, the protected accelerator and Substitution-Box (SBox) were designed, behaviorally verified with a high-level model, and implemented in Hardware-Descriptive Language (HDL).

The masking scheme in this work requires a Random Number Generator (RNG) in order to protect against side-channel attacks. Implementing a secure RNG is complex and has not been done in this work. One of the topics in NIST's roadmap for standardizing threshold implementations [19] is the RNG. Hopefully, a standard will be freely available online soon, but this work assumes that the accelerator is connected to an RNG that can deliver fresh randomness at the expected bandwidth.

1.2 Contributions

This work implements a first-order SCA secure AES accelerator for the CV32E40X core from the OpenHW group. The accelerator connects to the core using their novel XIF, which means that any core implementing this core can use the accelerator. The behavior of the accelerator follows the standardized behavior of the scalar-crypto ISE for RISC-V. The design process in this work consists of three stages: microarchitecture, high-level model, and HDL implementation. The microarchitecture can be found in Chapter 5, while the high-level models and HDL-implementations are available in the GitHub repositories presented in Appendix B. The implementation uses a 2-share Domain-Oriented Masking (DOM) scheme to protect against first-order SCA. The area of the resulting design is almost a factor of three larger and has a reduced throughput. However, the accelerator behavior complies with the scalar-crypto ISE standard, which means that there is no change in how the instructions are used. On the other hand, there is a reduction in throughput due to the implemented pipeline.

1.3 Structure of the Thesis

Before presenting the implementation of this work, a comprehensive background is given in Chapters 2, 3, and 4 with what is needed to understand the design and discussion. This also includes some descriptions of state-of-the-art technologies and their current developments. Afterward the design and implementations are presented in Chapter 5. The complete design is split into three stages: the accelerator, the SBox, and the XIF wrapper. Each of these stages are described in detail using their microarchitecture. In the end, an application using the implemented AES instructions is presented. Chapter 6 presents and compares the designs, and discusses various security issues that must be considered when integrating this accelerator.

Chapter 2

Advanced Encryption Standard

The Data Encryption Standard (DES) was the officially endorsed block cipher by the US government in 1977. Researchers scrutinized DES in a thorough research period, in which they found it to be vulnerable to analytical attacks [20]. One of the major criticisms by researchers was the small key size of only 56-bits. To prove that this was a significant weakness, RSA Laboratories launched three challenges where the goal of each challenge was to bypass DES encryption. The first challenge was solved after 140 days, and the cipher in the last challenge was solved in just 22 hours and 15 minutes [21]. At the same time, the US National Institute of Standards and Technology (NIST) called out for symmetric key block cipher candidates that they could declare as the new Advanced Encryption Standard (AES) [22].

2.1 The New Standard

NIST proposed a list of requirements that cipher candidates had to satisfy [23]:

- Mathematical proof of security for the cipher
- The developers must release the cipher without any license because the chosen cipher will be available worldwide and royalty-free.
- Algorithm should be efficient and able to run on an 8-bit Central Processing Unit (CPU).
- Should be suited for both software and hardware implementation.

NIST officially chose the Rijndael algorithm in October 2000 and expressed their motivation for this choice through a 116-page report [24]. It has since become one of the most widely used symmetric-key ciphers. To become AES, Rijndael had to undergo a rigorous standardization process, which reduced its flexibility by constraining the size of input parameters to limit the number of configurations. Figure 2.1 shows the standardized inputs and outputs. AES is an iterative block cipher operating on input blocks of 128-bits or 16-bytes. The secret key size is configurable to 128-, 192-, or 256-bits and is coupled to the number of iterations Nr, where a larger key requires more iterations.

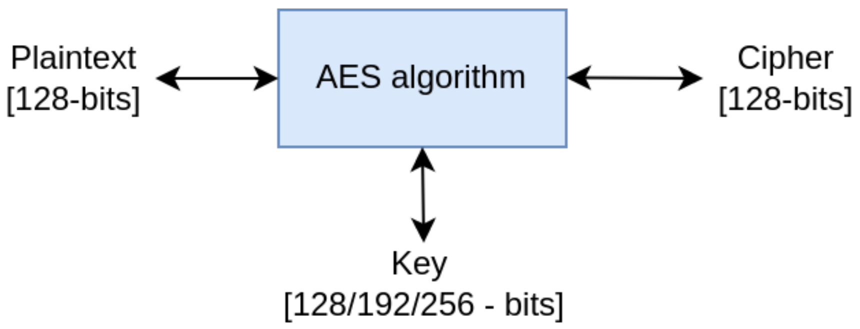

Figure 2.1: Input and output parameters of AES

2.2 Design of AES

Setting out to develop a new cipher is a large and complex challenge. It should be secure against known attack techniques, such as statistical analysis, while still having an efficient area and throughput. The developers of Rijndael used the wide trail strategy [4], a block cipher design strategy created by themselves when developing Rijndael. This strategy describes how to design and prove the security of new ciphers adequately.

2.2.1 Confusion and Diffusion

Claude Shannon presented in 1945 two essential properties in cryptography: confusion and diffusion [25]. Integrating these properties hardens a cipher against statistical attacks on its data. Confusion and diffusion are also further elaborated on in the Wide Trail Strategy for block ciphers and heavily impacted the design of Rijndael [22].

Confusion implies that the relation between the secret key and input ciphertext is complex. This property makes it difficult to deduce the secret key from a large set of encryption operations using the same key. Each bit in the ciphertext should depend on the whole secret key, meaning that changing a single bit in the key would result in a completely different output [25]. The second property, *diffusion*, implies there is a complex relationship between the input and output. A single bit flip in the input would result in an entirely different output. This property follows the strict avalanche criterion, which says that flipping any bit i in the input/output has a 50% probability of flipping bit j on the output/input [25]. In other words, diffusion eliminates any correlation between input and output data.

2.2.2 Number of Iterations

AES consists of multiple operations that introduce confusion and diffusion into the state. AES iterates over these operations 10, 12, or 14 rounds to provide sufficient security depending on if the input secret-key is 128-, 192-, or 256-bits, respectively. The developers of AES had to choose the round numbers to give an optimal performance while still providing sufficient security through diffusion and confusion [26]. There have been found shortcut attacks that are more efficient than exhaustive key-search attacks for AES with a key size of 128-bits for six rounds [27]. Using this as a baseline with the fact that two iterations of AES provide full-diffusion, the number of rounds was set to 10 for a 128-bit key [26]. Two additional iterations were added for each additional 32-bits in the secret key.

2.2.3 Finite Fields

The following section is based on [22].

A finite field is a constant set of elements. Operations in the field are constrained by certain properties [22], some of which are listed below. One of the important properties for cryptography is *closure* where the result of any operation is constrained within the field.

1 *Closure:* $\forall a, b \in F : a + b \in F$ and $a \cdot b \in F$

For all elements a and b in field F, the sum and product of the terms are constrained within the field

2 *Distributivity:* $\forall a, b, c \in F : (a + b) \cdot c = (a \cdot c) + (b \cdot c)$

For all elements a, b, and c in field F, the term outside the parenthesis can be multiplied by each factor.

3 *Commutative:* $\forall a, b \in F : a + b = b + a$

For all elements a and b in field F, the order of addition is trivial

4 *Neutral element*: $\exists 1 \in F, a \in F : a \times 1 = a$

There exists a multiplicative neutral element 1 in field F that when multiplied by a gives the same element a

5 *Multiplicative inverse*: $\forall a \neq 0 \in F, \exists a^{-1} \in F : a \cdot a^{-1} = 1$

For all elements a that are not 0 in field F, there exists an inverse of a in the field that results in the neutral element when multiplied by a

Finite fields are also called Galois Fields (GF) and are denoted as GF(p^n), where p is a prime number called the prime characteristic, and p^n is the field order. There are two types of finite fields: *prime fields*, where n = 1, and extension fields, where n > 1. Elements in a prime field are represented by integers, e.g., GF(2) that holds the elements {0, 1}. All operations in a prime field must follow the properties introduced above, which for a prime field means using modular arithmetic with the prime characteristic as the modulo. Table 2.1 shows the results from addition, subtraction, and multiplication in GF(2). It is observable that addition and subtraction give the same result, and their calculation is the same as an Exclusive-Or (XOR) operation. While multiplication is the same as an AND-operation.

+-	**0**	**1**
0	0	1
1	1	0

x	**0**	**1**
0	0	0
1	0	1

Table 2.1: Addition results(left) and multiplication results(right) in GF(2)

Straightforward modular arithmetic for extension fields does not follow the required properties for finite fields. Instead, different representations, or bases, are utilized. A typical representation for extension fields is polynomial representation, shown in Equation (2.1) where the coefficients b_i are in the prime field GF(p) and the polynomial degree is n. Thus, the Galois Field GF(2^2) contains the elements {0, 1, x, x+1}. Operations on extension fields still use modular arithmetic, but now the modulo is an irreducible polynomial p(x). There can be multiple irreducible polynomials for a finite field, and it must be specified for a given field since operations are dependent on the irreducible polynomial. Hence, the operations can differ from one field to another if the irreducible polynomial is different.

$$b(x) = \sum_{i=0}^{n-1} b_i x^i = b_{n-1}x^{n-1} + ... + b_2x^2 + b_1x + b_0 \tag{2.1}$$

Addition and multiplication using polynomial basis follow straightforward standard polynomial operations, but the coefficients are in the prime field. Thus, operations on individual coefficients of a polynomial in a field GF(p^n) follow the properties of the prime field GF(p). Addition or subtraction of polynomials for an extension field with the prime characteristic 2 is modeled by XORing the coefficients of the same order as was shown in Table 2.1. Examples of addition and multiplication is given in Equations (2.2a) and (2.2b), respectively, for the field GF(2^2) with the irreducible polynomial p(x) $= x^2 + x + 1$, $a = x + 1$, and $b = x + 1$

$$\begin{aligned} c &= a + b \mod p(x) \\ &= (x+1) + (x+1) \mod x^2+x+1 \\ &= (x+x) + (1+1) \mod x^2+x+1 \\ &= 0 \end{aligned}$$

(a)

$$\begin{aligned} c &= a \times b \mod p(x) \\ &= (x+1) \times (x+1) \mod x^2+x+1 \\ &= x^2 + (x+x) + 1 \mod x^2+x+1 \\ &= x^2 + 1 \mod x^2+x+1 \\ &= x \mod x^2+x+1 \end{aligned}$$

(b)

Equation 2.2: Polynomial addition (a) and multiplication (b)

Any element represented with a polynomial basis for an extension field using prime characteristic $p = 2$ is a bit-vector. The elements in GF(2^8) represent a byte, as represented in Equation (2.3). The finite field used in AES if $GF(2^8)$ with the irreducible polynomial $p(x) = x^8 + x^4 + x^3 + x + 1$.

$$b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \tag{2.3}$$

Another widely used representation is the Normal basis. The coefficients in a Normal basis still represent the bits of the element, but the terms follow the form $\{\beta, \beta^p, \beta^{2p}, ..., \beta^{(n-1)p}\}$ [28]. A normal basis representation in GF(2^8) is shown in Equation 2.4.

$$b(x) = b_7\beta^{128} + b_6\beta^{64} + b_5\beta^{32} + b_4\beta^{16} + b_3\beta^8 + b_2\beta^4 + b_1\beta + b_0 \tag{2.4}$$

As was shown in Table 2.1, operations in GF(2) can be represented by boolean circuitry. Operations in lower-order fields are typically more straightforward than in fields with higher orders. An extension field can be converted to a composite field, in which an element can be split into smaller elements and utilize subfield operations. The conversion is possible since finite fields are isomorphic, meaning that finite fields have a linear mapping that can be used for such a conversion [22]. E.g., there exists a linear mapping that maps a value in GF(2^8) to GF($((2^4)^2)$), where an 8-bit element can be represented as two 4-bit nibbles. The equation below shows how a composite field element is represented with a polynomial in Equation

(2.5a) and normal basis in Equation (2.5b), where A is the isomorphically transformed 8-bit value, a_h is the higher nibble, and a_l is the lower nibble.

$$A = a_h x + a_l \tag{2.5a}$$

$$A = a_h \beta^{16} + a_l \beta \tag{2.5b}$$

Elements in GF(2^8) can also be mapped to the composite field $GF((((2^2)^2)^2)$; wherein an 8-bit element can be split into elements in the prime field [29]. I.e., operations on an 8-bit element can utilize the simple operations in GF(2). Each level of decomposition requires its own irreducible polynomial. Equation 2.6 shows a generic irreducible polynomial, where τ is called the *Trace*, and μ is called the *Norm*.

$$p(x) = x^2 + \tau x + \mu \tag{2.6}$$

The irreducible polynomials for the composite field $GF(((2^2)^2)^2)$ are shown in Equation (2.7), where (T, τ) are the Traces and (N, v) are the Norms. In GF(2), there is only one irreducible polynomial where both the Trace and Norm are 1. Thus, the other polynomials can not have 1 for both the Trace and Norm. Traces are usually set to 1 for efficient implementations in hardware [30]. Then the Norm must be chosen so that the resulting polynomial is irreducible for the composite field.

$$\begin{aligned} GF(2^2)/GF(2) &: p(w) = w^2 + w + 1 \\ GF(2^4)/GF(2^2) &: s(z) = z^2 + Tz + N \\ GF(2^8)/GF(2^4) &: p(y) = y^2 + \tau y + v \end{aligned} \tag{2.7}$$

2.3 AES Algorithm

The AES encryption and decryption algorithm are shown in Figure 2.2. AES consists of four operations: AddRoundKey, Byte Substitution, ShiftRow, and MixColumns, all of which have an inverse used for decryption. The colored area in the figures indicates the operations executed iteratively for Nr rounds, where Nr is 10, 12, or 14 rounds, which is configured through the key size. As observable in the figure, decryption is the reversal of encryption. The MixColumns operation must be skipped in the last encryption iteration to make the structures similar for both procedures. Each iteration requires a 128-bit round key, which is generated from the secret key using the KeyScheduler algorithm described in Section 2.3.5

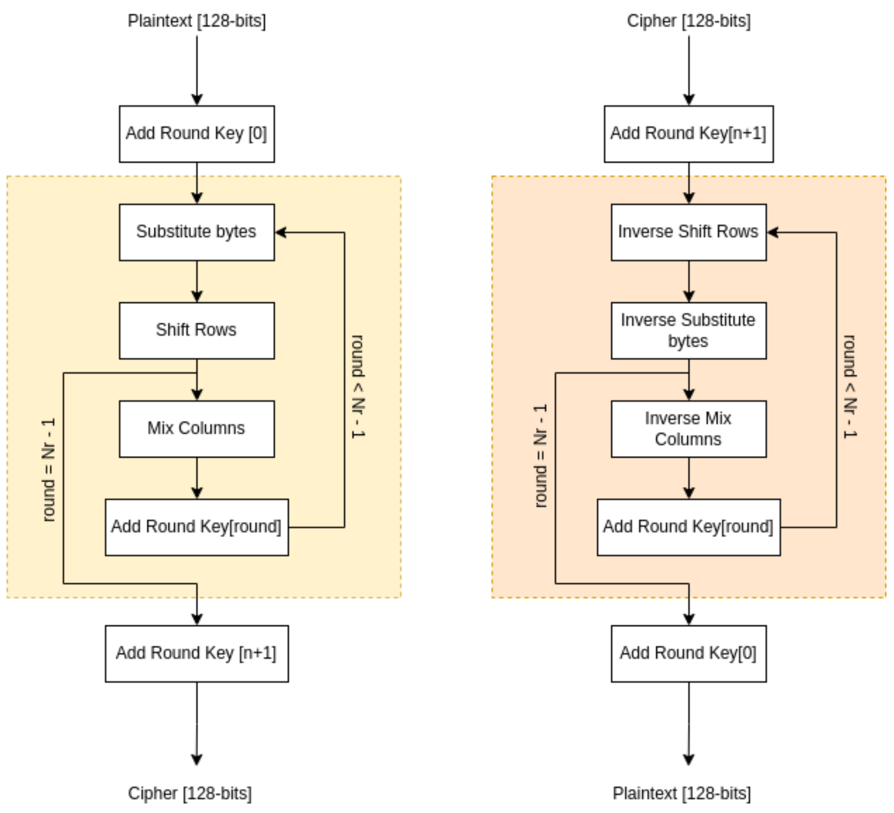

Figure 2.2: AES encryption and decryption algorithms

The 128-bit input block, plaintext or ciphertext, is split into 16-bytes $B_0 B_1 B ... B_{15}$ and placed in a column-major order in a matrix as shown in Figure 2.3. This matrix is referred to as the state matrix when input into the algorithm. The various operations in the algorithm operate on individual bytes, rows, or columns.

$$\begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{bmatrix}$$

Figure 2.3: Byte order in state matrix for input and output

2.3.1 AddRoundKey

In the AddRoundKey operation, a round key of 128-bits is added to the state matrix as shown in Equation (2.8). Each iteration of this operation uses a different round key; thus, each procedure requires $Nr + 1$ round keys. The inverse operation for AddRoundKey is to subtract the round keys from the state in the reverse order of encryption. As described in Section 2.2.3, addition and subtraction for an extension field with prime characteristic 2 is the same as an XOR operation. This characteristic means that AddRoundKey is the same operation for encryption and decryption [22].

$$\begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{bmatrix} \oplus \begin{bmatrix} RK_0 & RK_4 & RK_8 & RK_{12} \\ RK_1 & RK_5 & RK_9 & RK_{13} \\ RK_2 & RK_6 & RK_{10} & RK_{14} \\ RK_3 & RK_7 & RK_{11} & RK_{15} \end{bmatrix} \tag{2.8}$$

2.3.2 Substitution-Box

One of the more complex operations of AES is the Substitution-Box (SBox) transformation. This step operates on individual bytes and adds diffusion through non-linearity [22]. The non-linearity dissipates statistical correlations in the input and output to protect against linear cryptanalysis. The SBox transformation of AES consists of two operations: an inversion in GF(2^8) and an affine transformation, as shown in Figure 2.4.

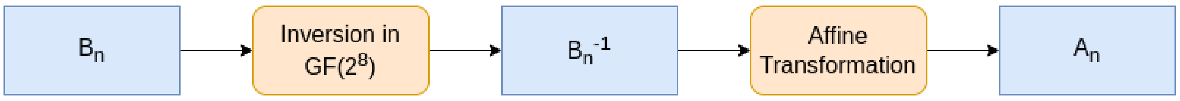

Figure 2.4: AES SBox split into two sub operations

The inversion alone has a simple algebraic expression, which statistical attacks could abuse. In combination with the inversion, the affine transformation results in a more complicated expression while still keeping the non-linear properties introduced by inversion. An *affine transformation* is a geometric transformation that maps the input onto itself, such as scaling, rotations, and translation. Hence, an affine transformation modifies the input while keeping the geometric structure [31]. The affine transformation used in AES encryption is defined in Equation 2.9a where a_n and b_n are the bits of the input and output byte. The inverse, which is used in decryption, is shown in Equation 2.9b.

$$aff = \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

(a)

$$aff^{-1} = \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

(b)

Equation 2.9: AES Forward affine transformation (a) and inverse affine transformation (b)

Since GF(2^8) is a finite set of 256 elements and the affine transformation follows the constant multiplication and addition in Equation 2.9a, it is feasible to create a precalculated lookup table as seen in Table 2.2. Using a lookup table is fast and simple in software, but this would require 256x2 bytes of memory and additional circuitry for addressing and fetching memory elements in hardware. These demands are too high for some applications, such as smart cards, and in those cases, it might be more efficient to implement the transformation itself [29][30].

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

Table 2.2: Precalculated SBox lookup where the leftmost column represents the lower nibble and the uppermost row represents the higher nibble

Canright presented a highly optimized implementation using a tower of fields approach [30]. He found that using a normal basis resulted in one of the more area-efficient implementations for hardware. Furthermore, he found that composite field operations were more efficient for the inversion but not linear operations such as the affine transformation or the rest of AES. Hence, his paper focuses on an optimized version of the inversion operation in $GF(2^8)$. The inversion starts with an initial isomorphic transformation that converts an element into the composite field $GF(((2^2)^2)^2)$, after which subfield operations are used to perform the inversion. In the end, the result is mapped back into GF(2^8) and outputs the inverted value. *Joan Boyar and Rene Peralta* also used the tower field approach to implement one of the smallest SBoxes using only 127 gates at a depth of 16 in [32].

Figure 2.5 depicts the inversion in the composite field $GF((2^4)^2)$ using a normal basis. Each operation can be converted further down to a lower subfield, which is done for area-efficient hardware implementations. As observable in the figure, this inversion consists of two additions, one square-and-scale, three $GF(2^4)$ multiplications, and a $GF(2^4)$ inversion. The proof for this inversion is derived in Equation A.3 in Appendix A. Additionally, the derivations for each subcomponent down to the lowest composite field can also be found in Appendix A.

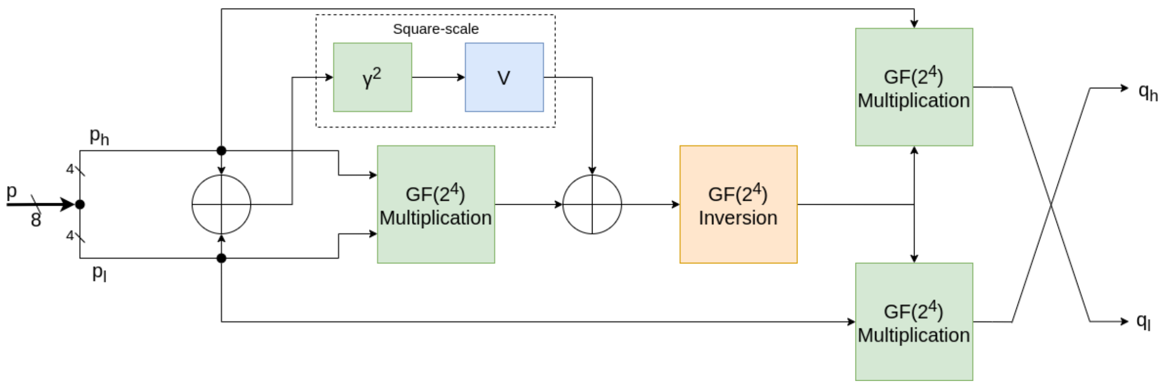

Figure 2.5: $GF(2^8)/GF(2^4)$ composite field inversion

2.3.3 ShiftRows

As implied by the name, this step shifts the rows in the state matrix. It is one of the steps used to create diffusion in the algorithm [22]. Each row of the state matrix is left-shifted by the index of its respective row, starting at the top with index zero. Elements that extend outside the matrix are inserted back on the right side, as shown in Equation (2.10). During decryption, the rows are right-shifted, reversing the operation from encryption.

$$\begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{bmatrix} \rightarrow \begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_5 & B_9 & B_{13} & B_1 \\ B_{10} & B_{14} & B_2 & B_6 \\ B_{15} & B_3 & B_7 & B_{11} \end{bmatrix} \tag{2.10}$$

2.3.4 MixColumns

MixColumns is another simple linear operation implemented to introduce more diffusion [22]. This operation inputs the four bytes of a column and outputs another four bytes, following matrix multiplication in Equation 2.11a. Each output byte of this step consists of four scalar multiplications and three additions, which are still in GF(2^8). The inverse operation for decryption is shown in Equation 2.11b.

$$MC = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 1 & 2 \\ 1 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 \\ 2 & 3 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

(a)

$$MC^{-1} = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} 0B & 0D & 09 & 0E \\ 0D & 09 & 0E & 0B \\ 09 & 0E & 0B & 0D \\ 0E & 0B & 0D & 09 \end{bmatrix} \times \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

(b)

Equation 2.11: AES MixColumns(a) and inverse MixColumns(b)

One of the typical approaches to calculating the scalar multiplication is by using the *xtime* function [2], as shown in Listing 1. The xtime function creates an addition chain of xtime2, which accumulates into the scaled input. The xtime2 function doubles the input byte. If the input byte's most significant bit is 1, the doubling will increase the value outside the field.

Thus, a modular reduction using the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$, 0x11b in hex, keeps the result within the field.

```
uint8_t xtime2(uint8_t byte){
        bool  = (byte&0x80) > 0;
        return (byte << 1) ^ (msb ? 0x1b : 0);
}

void xtime(uint8_t byte, uint8_t scalar){
        return  ( scalar&0x1 ? byte : 0 )) ^ \
                ( scalar&0x2 ? xtime2(byte) : 0 ) ^ \
                ( scalar&0x4 ? xtime2(xtime2(byte)) : 0 ^ \
                ( scalar&0x8 ? xtime2(xtime2(xtime2(byte))) : 0);
}
```

Listing 1: xtime function in C for performing scalar multiplication in GF(2^8)

2.3.5 KeyScheduler

As previously mentioned, $Nr + 1$ round keys of 128-bits are needed for each encryption or decryption. However, the input key for AES is a single key of 128-, 192-, or 256-bits. A Key Schedule algorithm uses the secret key to derive the correct number of round keys. This algorithm operates on 32-bit words W. The number of input blocks N is the number of 32-bit words in the input key. Using these parameters, the words W_i are generated using Equation 2.12. A visual presentation is shown in Figure 2.6 for the first 16-bytes of an AES 256-bit key.

$$W_i = \begin{cases} K_i, & \text{if } i < N. \\ W_{i-N} \oplus SubWord(RotWord(W_{i-1})) \oplus rcon_{i/N}, & \text{if } i \geq N \text{ and } i = 0 \mod N \\ W_{i-N} \oplus SubWord(W_{i-1}), & i \geq N, N > 6, \text{and } i = 4 \mod N \\ W_{i-N} \oplus W_{i-1}, & otherwise \end{cases}$$

Equation 2.12: KeyScheduler word algorithm

Three operations transform every N word: SubWord, RotWord, and rcon. SubWord is the same operations as the SBox for encryption, where each byte in the word is transformed by taking the inverse in GF(2^8) and then performing the affine transformation. RotWord left shifts the bytes in the word by one, where the byte that falls off is inserted as the least significant byte, just as in ShiftRows. In the end, a round constant is added to the word. The round constant, rcon, is generated using Equation 2.13 and shifted up by three bytes before it is added to the key. The $(N_r + 1) \times 4$ generated words are sequentially split into groups of four or 128-bits as is the required round key length.

$$rc_i = \begin{cases} 1, & \text{if i = 1} \\ 2 \cdot rc_{i-1}, & \text{if} > 1 \text{ and } rc_{i-1} < 0x80 \\ (2 \cdot rc_{i-1}) \oplus 0x11B, & \text{if} > 1 \text{ and } rc_{i-1} > 0x80 \end{cases}$$

Equation 2.13: Generation of round constants for round keys

A visual example is given for AES-256 in Figure 2.6, where the first 16-words are generated from the input key. For a 256-bit key there are N = 8 words; hence, every eight word propagate through the three operations, and very four words after the first eight, go through the SBox.

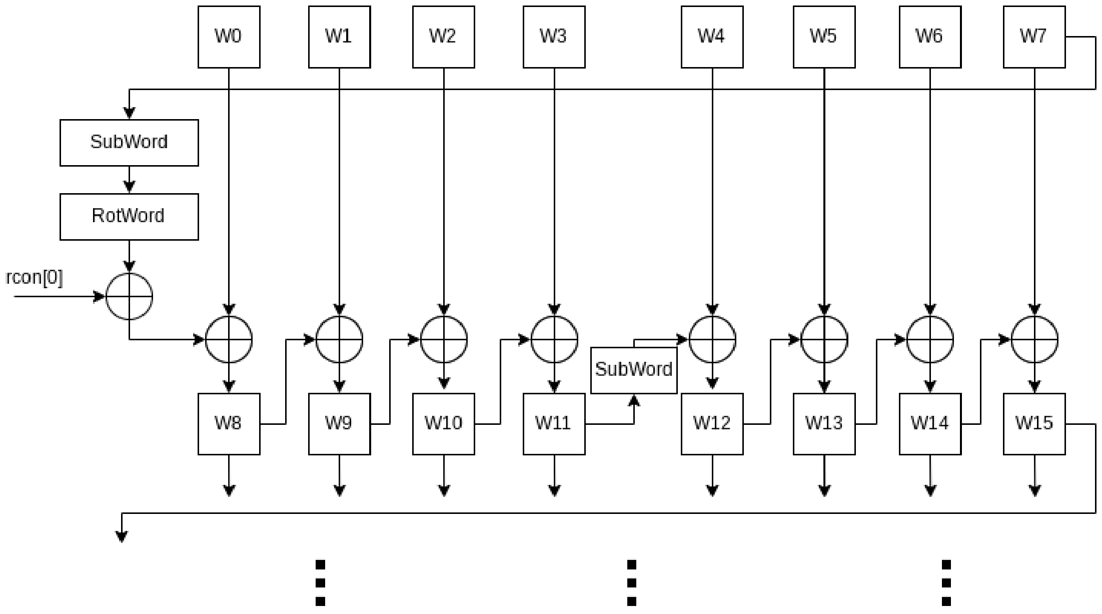

Figure 2.6: Key Schedule algorithm for a 256-bit input key

2.3.6 Equivalent Decryption

Reversing the encryption procedure, and using the inverse operations, results in the decryption algorithm. However, AES's properties also enable an equivalent decryption procedure that could be more efficient for certain implementations [2]. By first performing the Inverse MixColumns on all round-keys, except for the first and last, it is possible to swap the order of the Inverse MixColumns and AddRoundKey in the decryption process, as shown on the right side in Figure 2.7. Comparing this to the encryption process on the left side shows how this equivalent procedure implements an identical path to the encryption path, making it possible to share most of the resources for both processes, which is area efficient for hardware implementations. However, this reduces the overall efficiency due to the additional work to create new decryption round keys.

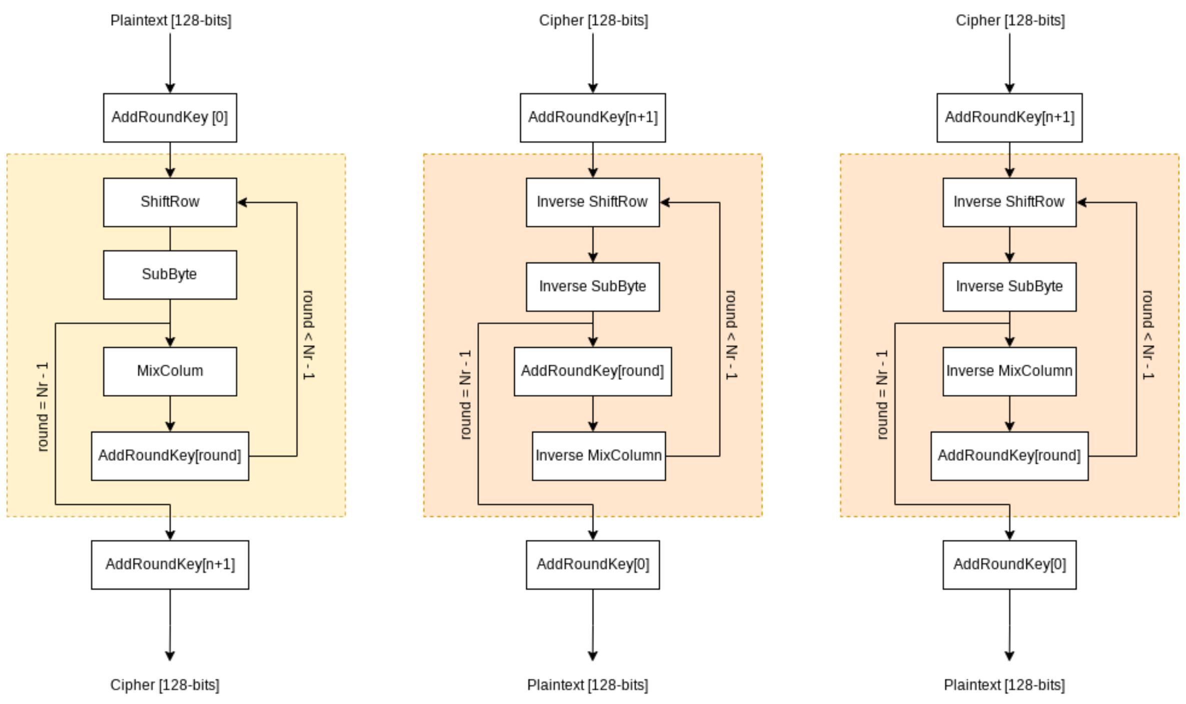

Figure 2.7: AES encryption(left), decryption (middle), and equivalent AES decryption (right) where AddRoundKey requires round keys that have gone through Inverse MixColumns.

Chapter 3

Information leakage and countermeasures

The smallest key for Advanced Encryption Standard (AES) is 128-bits, which gives more than 10^{38} combinations for a single key. With modern ciphers utilizing such large keys, exhaustive key-search, also known as a brute-force attack, is infeasible even with modern computers. Instead, researchers started investigating the hardware of cipher implementations. The research discovered information leakage through physically measurable parameters such as power consumption [5], timing [33], temperature [34], and electromagnetic emissions [35]. These physical parameters are referred to as side-channels. An attack that collects side-channel data and analysis it to deduce secret information is called a Side-Channel Attack (SCA). In 1999 Paul Kocher and al. presented a simple and differential power analysis technique that could be used to acquire secret data from physical attacks on hardware [5]. More complex techniques have since been developed. Consequently, an effort had to be put into creating countermeasures against these powerful attack vectors.

3.1 Preliminaries

The most prominent SCAs utilize data from power consumption. This section will present some preliminaries needed to understand the attack techniques and countermeasures against them.

3.1.1 Power-Consumption

The following paragraphs are based on [36].

The total power consumption of a device can be split into two contributions: static and dynamic. *Static power consumption* is a constant power consumption caused by leakages, while dynamic power consumption is the power consumed by device activity. The consumption is dependent on the technology that the circuit is built on. One of the more typical technologies uses Complementary Metal-Oxide Semiconductors (CMOS) to build logic gates. CMOS logic gates are designed using a pMOS pull-up network and an nMOS pull-down network. The total power consumption of a digital circuit is the accumulated static and dynamic power of all the CMOS gates. The static power of a CMOS gate is induced by a leakage current through the transistors and the voltage amplitude of the power supply, as seen in Equation 3.1. The resulting product is a small contribution at an individual gate level; however, since modern circuits can consist of millions of these gates, the contribution accumulates to a significant factor

$$P_{static} = V_{DD} \times I_{leak} \tag{3.1}$$

In a CMOS logic gate, the pull-up network charges the capacitive load on the output when enabled, while the pull-down network discharges the load. The gate only sources power when the output changes from '0' to '1'. Thus, a gate must undergo a complete transition to consume power. The dynamic power consumption of a CMOS gate is defined in Equation 3.2, where α is the activity factor of the gate, f is the clock frequency, C_{load} is the capacitive load, and V_{DD} is the power supply. The activity factor describes how often the output of the gate toggles from low to high on average per clock tick. Thus, a computationally heavy unit with a high activity requires significantly more power than a unit with less activity.

$$P_{dyn} = \alpha f C_{load} V_{DD}^2 \tag{3.2}$$

3.1.2 Glitching

Imperfections in physical implementations can often lead to behaviors unexpected by the designer. Signals propagating through digital circuitry can be delayed by the length of a path and the number and type of logic gate it must propagate through. For a logic gate dependent on two inputs, one signal arriving before the other could cause an unwanted change in the output until the second input arrives. The unwanted change is called a *glitch* and is a prevalent behavior in digital circuitry. So much that it contributes a total of 20%-70% of overall power consumption [37]. Figure 3.1 depicts how a glitch could be produced for an AND-gate, where Signal 1 is initially low and signal 2 high. Signal 2 arrives shortly after signal 1, but the output of the gate has already toggled, resulting in a glitch propagating through the rest of the circuitry.

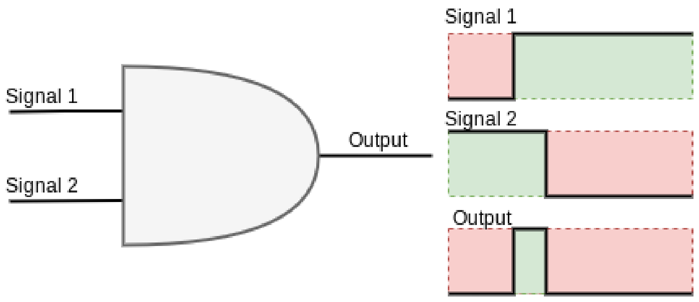

Figure 3.1: How a glitch is produced for an AND-gate

3.2 Physical Attacks

Having physical access to a device enables more attack techniques. Physical attacks are usually split into two categories: active and passive. In active attacks, an adversary attempts to affect the behavior of a circuit by directly tampering with it. *Fault injection* is a typical example in which an adversary could modify the clock frequency or source voltage in a processor to make it "skip" instructions [38]. Passive attacks measure the side-channels of a device and do not require any modifications to the hardware. SCAs have become prominent, and the most straightforward analysis techniques have online guides and purchasable devices that can be used for that purpose [6].

3.2.1 Side-channel Attacks

The power consumption of a device is strongly correlated to the activity in a circuit. The analysis techniques presented by *Kocher et Al.* in 1998 use this to deduce information about the underlying algorithm implemented in hardware or use the statistical correlation between a guess and the power consumption to deduce secret values. Three of the most known approaches are simple, differential, and correlation power analysis.

3.2.2 Simple Power Analysis

The most straightforward analysis approach is the Simple Power Analysis (SPA). In this approach, a single trace of the power consumption is used to deduce information about the algorithm that is executed. The power consumption reflects how active the circuit is, as described by the dynamic power from Section 3.1.1. In a hardware implementation consisting of multiple stages with different complexities, the power consumption will outline a characteristic of the operation executed in the stage. If there are 10, 12, or 14 successive peaks or intervals in the power consumption, one could confidently guess that the underlying algorithm is AES [39]. Figure 3.2 shows the power consumption for a device executing AES with a 128-bit key. There are nine similar intervals with a shorter one at the end. These intervals represent the first nine rounds of AES-128, with the last round being shorter due to skipping MixColumns.

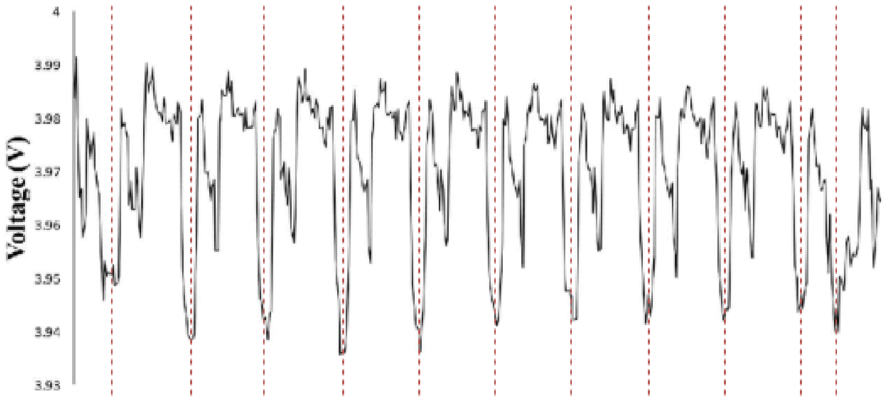

Figure 3.2: Power consumption of a device encrypting using AES-128. The figure is a modified version from [40].

3.2.3 Differential Power Analysis

Differential Power Analysis (DPA) uses the principle that flipping a bit from low to high has a higher power consumption than the opposite [5], which is the case for the CMOS logic gates. CMOS gates source power when the output is high and drain power when low. The variation for flipping a single bit is tiny and overshadowed by noise. However, this noise could be reduced by averaging a large set of traces [5].

In the difference of means DPA, one attempts to guess the constant secret key one byte at a time [39]. In short, this is done by correlating the power consumption and self-generated key guesses. First, one would collect a large set of traces T_i coupled with the corresponding output cipher C_i. These traces are split into two groups by a selection function $D(K_n, C_i)$, where K_n is a one-byte key-guess, and C_i is one of the collected output ciphertexts. The selection function reverses the operations in the last round of AES using key-guess K_n back to the output of the Substitution-Box (SBox) using the inverse operations. One of the bits in the

SBox output guess is selected and used to group the traces. Typically, the least significant bit is chosen, which gives the groupings: $Group_0$ that contains all traces where the least significant bit is zero and $Group_1$ that contains the traces where the least significant bit is one.

For each key-guess a trace for the difference ΔD_n between the average of $Group_1$ and $Group_0$ is calculated, as seen in Equation (3.3).

$$\Delta D_n = \frac{\sum^i D(K_n, C_i) \times T_i}{\sum^i D(K_n, C_i)} - \frac{\sum^i (1 - D(K_n, C_i)) \times T_i}{\sum^i (1 - D(K_n, C_i))} \tag{3.3}$$

The trace with the correct K_n will have a significant peak in the corresponding ΔD_n trace since the bit that was used in the selection function will be correct for all traces, meaning that there is a strong correlation between the traces. Figure 3.3 gives an example of how the traces for ΔD_n could be. The uppermost two lines have no significant correlation, while the lowermost one has a significant peak. Thus, the key guess for this trace is probably correct.

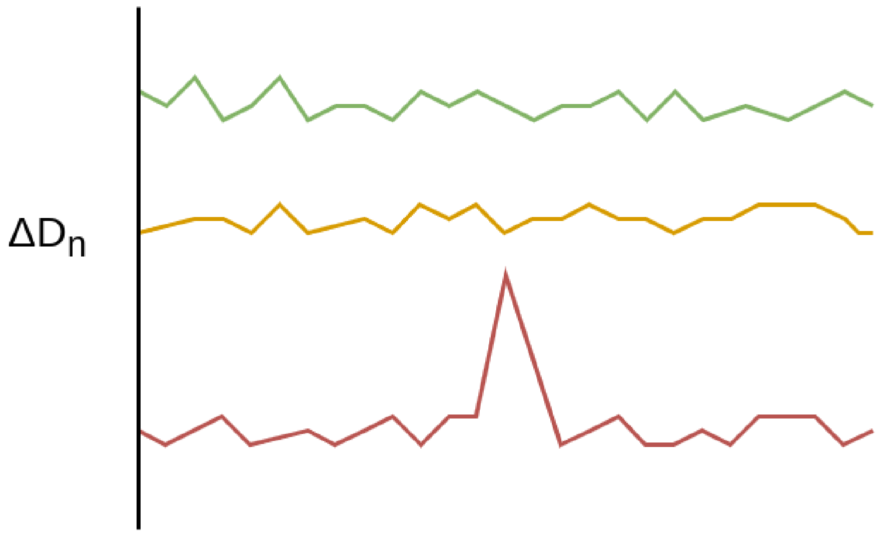

Figure 3.3: Example correlation plot for an arbitrary difference of means DPA where each trace represent the correlation at time/sample x for key guess n.

3.2.4 Correlation Power Analysis

Correlation Power Analysis (CPA) uses the same basis as DPA, but instead of using a selection function, it uses a power model to find correlation [41]. The most typical model is a hamming weight model, where the power consumption of a bit-vector increases proportionally to the number of high bits. E.g., the integer 63 is 0011 1111 in binary with a hamming weight of 6, while the integer 64 is 0100 0000 in binary with a hamming weight of 1.

This approach is often used for implementations that operate on a single byte at a time, such as accelerators implemented in Central Processing Unit (CPU). CPA is typically used when the adversary can control the input. The procedure for CPA is that for each power trace T_i the algorithm receives a known random input byte PT_i. The adversary computes the AddRoundKey with a key-guess K_n and applies the SBox transformation. The hamming weight of the SBox output is coupled to the corresponding trace and key-guess. This computation is done for all 256 key guesses. An extensive set of traces and hamming weights are collected, and in the end, the correlation coefficient for each key-guess and the traces are found [39]. Since there is a linear relation between the power consumption and the hamming weight, there should be a point in the traces where one of the hamming weight sets has a strong correlation [41].

Equation (3.4) shows how to calculate the correlation between the measured power traces $T_i$and the hamming weights H_i, where $\bar{T}$ and $\bar{H}$ are mean values. Figure 3.4 gives a visual representation of the data used in a CPA. The plot on the left depicts five plotted power consumption over time x, and the matrix on the right consists of arbitrary hamming weights for key guess k. A correlation coefficient $r_{i,k}$ is found for the set of hamming weights for each key guess k at each trace point j. If any correlation is found, one of the correlation traces r_k will have a significant peak at a point i.

$$r_{i,k} = \frac{\sum_i (T_{i,j} - \bar{T})(H_{i,k} - \bar{H})}{\sqrt{\sum_i (T_{i,j} - \bar{T})^2 \sum_i (H_{i,k} - \bar{H})^2}} \tag{3.4}$$

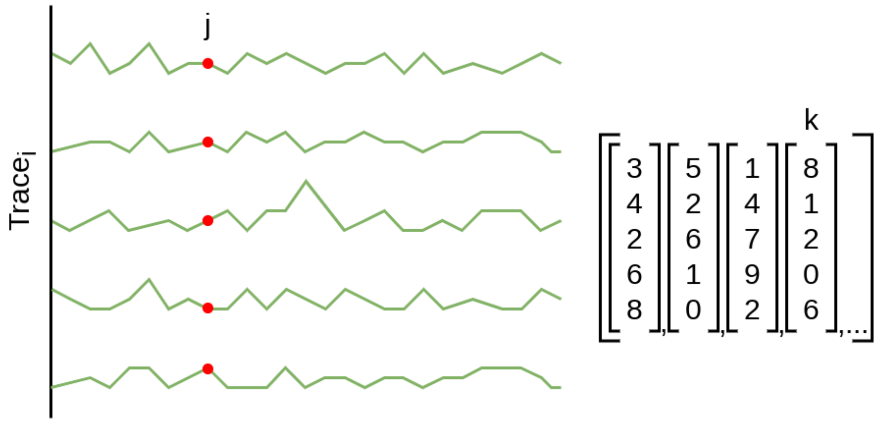

Figure 3.4: Visual representation of an arbitrary CPA attack, where the plot depicts the collected power traces over time x, and the matrix represent the hamming weight for the corresponding key guess

3.2.5 Higher-Order Attacks

First-order SCA attacks are, in essence, simple because they only operate on one data point per trace at a time. However, a typical countermeasure is a share-splitting scheme where the input data is split into multiple parts called *shares*, where all shares are required to get the unshared value. This is further described in Section 3.3.1.

Higher-order attacks combine data from multiple data points to determine the unshared value [5], which increases the complexity significantly. These attacks were first proposed by *Kocher et al.* in their original SCA article [5], and since then DPA has been extended to Higher-Order Power Analysis (HO-DPA) with some successful practical usecases [42]. A further development of HO-DPA is Multivariate Information Analysis (MMIA) [43], which is a more generic approach compared to HO-DPA. Higher-order attacks attempt to find correlations in a combination of multiple data points, but finding the correct data points is one of the major challenges in these techniques. The state-of-the-art techniques are adopting machine learning technologies to perform SCAs [44] or to deduce the value of the mask [45].

3.3 Countermeasures

As a consequence of side-channel attacks, straightforward hardware implementations of ciphers cannot be considered secure. Researchers have since attempted to develop various countermeasures against such physical attacks. Countermeasures that gain popularity are thoroughly scrutinized by the research community to make sure that they are sound.

Some of the initial novel measures included adding random delays or noise in the execution, making the alignment of the sets difficult. However, this was deemed to be unsuitable countermeasures since signal processing techniques could reduce their impact [46], [8], [47]. Instead, researchers started developing measures that could be mathematically proven to be sound.

3.3.1 Share-Splitting and Masking

Two of the base concepts used as counters for AES are the secret-sharing scheme and masking. In the secret-sharing scheme, a secret value is split into multiple shares. Combining the shares results in the original secret value, but a single share, or any other combination of shares, does not give enough information to deduce the original value [48]. Operating on a single share does not leak any information through side-channels. For AES, the shares are created by adding a random number to the input; one share is the sum of the random number(s) and the input, and the other share(s) are the random number. Equation 3.5 shows how the shares are created and the generic idea on how to extend it for more shares. The values $mask_0$ and $mask_1$ are randomly generated numbers. Adding a random number to the original input is a technique called masking. Intermediate values are randomized when operating on a masked value, removing the correlation on the side-channels. This random value is added at the beginning and removed at the end to reveal the correct result.

$$\begin{aligned} Share_0 &= input + mask_0 + mask_1 \\ Share_1 &= mask_0 \\ Share_2 &= mask_1 \end{aligned}$$

Equation 3.5: Share splitting scheme for three shares

Algorithms with only linear operations can operate on the shares in parallel and get the correct result when recombining them. However, non-linear operations, such as the SBox transformation of AES, are more complicated. The SBox has, as a consequence, been one of the main focus areas for researchers. Multiple techniques have been developed, where some were proven to be insecure later.

Additive and multiplicative masking techniques are two of the more straightforward approaches early developed for the SBox. In additive masking, a mask is added to the value and removed at the end. This masking technique is simple for linear operations but not for non-linear operations. Instead, multiplicative masking was used for non-linear operations. Algorithms integrating both types can use a combination of additive and multiplicative masking. During the transition from one type of masking to another, it is important to make sure that the signal is not unmasked at any step. Equation 3.6 shows a modified flow of AES' inversion using multiplicative masking [49].

$$\begin{aligned}
(x+m) \quad & \mid \times m' \\
(x+m)m' \quad & \mid +(m \times m') \\
x+m' \quad & \\
(x+m')^{-1} \quad & \\
x^{-1}m'^{-1} \quad & \mid +(m \times m'^{-1}) \\
(x^{-1}+m) \times m'^{-1} \quad & \mid \times m' \\
x^{-1}+m \quad &
\end{aligned}$$

Equation 3.6: The transition from additive masking at the input to multiplicative masking resulting in an additive mask

These masking schemes were developed with the assumption that gates only switch once per clock tick. However, glitches are prevalent in digital circuitry and contribute significantly to the dynamic power consumption. *Stefan M. et al.* investigated the current masked implementations and proved how they were all susceptible to first-order DPA attacks due to glitches [50].

3.3.2 Threshold implementation

Nikova et al presented a *Threshold Implementation (TI)* scheme for AES that does not leak information, even in the presence of glitches [9]. Their initial observation was that, in a shared implementation, linear operations do not leak information because each operation only depend on a single share. On the other hand, non-linear operations must combine shares to get a correct result when recombining them. Intermediate signals depending on all shares can cause leakages through glitches. In their proposal, they formalized three properties that must be fulfilled for a secure implementation:

1. Correctness
2. Non-completeness
3. Uniformity

The first property entails that the result of recombining the shares must give the same result as if the original implementation was not shared at all.

The second property means that each function should be independent of at least one input share. A linear operation fulfills this property, as it operates on a single share. However, making a non-linear operation, such as multiplication, non-complete, requires additional effort. A non-complete implementation of multiplying factors A and B requires a minimum of three shares since it is impossible to derive two functions for the operations that fulfill both correctness and non-completeness. Equation 3.7 shows a possible derivation of three non-complete functions for a three share multiplication with the terms A and B. The functions F_0, F_1, and F_2 are all independent on one of the input shares. Combining the three functions result in $F = A \times B$.

$$
\begin{aligned}
F_0 &= A_1B_1 + A_2B_1 + A_1B_2 \\
F_1 &= A_2B_2 + A_2B_0 + A_0B_2 \\
F_2 &= A_0B_0 + A_0B_1 + A_1B_0
\end{aligned}
$$

Equation 3.7: Three share multiplication for threshold implementation

The last property implies that the input and output of a function must have a uniform distribution. In a non-uniform distribution, there is a higher probability of getting one value at the output than another, and if a signal propagates through a chain of these functions with a non-uniform distribution, the peak of the skewed distribution will increase. No generic functions have been found to give uniformity. Hence, the functions presented in Equation 3.7, which fulfill both properties one and two, are not secure implementations. A remasking scheme adds fresh randomness to each of the functions to make them uniform [51]. A remasked uniform version of the multiplier is shown in Equation 3.8 where M_n are randomly generated values.

$$
\begin{aligned}
F_0 &= A_1B_1 + A_2B_1 + A_1B_2 + M_0 \\
F_1 &= A_2B_2 + A_2B_0 + A_0B_2 + M_1 \\
F_2 &= A_0B_0 + A_0B_1 + A_1B_0 + M_0 + M_1
\end{aligned}
$$

Equation 3.8: A uniform three share multiplication for threshold implementation

Since the first research paper about TI by *Nikova et al.*, a handful of variations have been developed, some optimizing for speed, area, or the number of required fresh random bits. Due to the importance of using masking schemes to protect against powerful side-channel attacks, National Institute of Standards and Technology (NIST) announced a new project to standardize threshold schemes for ciphers [13]. In July 2020, NIST released a roadmap with requirements for the schemes [19].

3.3.3 Domain Oriented Masking

Another TI variation is the Domain-Oriented Masking (DOM) scheme presented by Hannes Groß in [10]. A DOM implementation requires $d + 1$ shares to be secure against *d-th* order SCAs. Each share forms its own domain. While TI schemes usually focus on the different functions that make up the non-linear operations, DOM focuses on not combining shares of different domains. As in a standard TI implementation, linear operations are not problematic, and non-linear operations such as multiplications are more complicated. Multiplying the shares of the two terms A and B, where A_0 and A_1 are the shares for A and B_0 and B_1 are the shares for B, gives the product terms: A_0B_0, A_0B_1, A_1B_0, and A_1B_1. These terms are split into two groups: *inner-domain* terms and *cross-domain* terms. *Inner-domain* terms only have shares within one domain, i.e., A_0B_0 and A_1B_1. These terms do not cause any leakages as they are not dependent on shares from other domains. *Cross-domain* terms combine shares from different domains, i.e., A_0B_1 and A_1B_0. These are critical terms and can cause leakages when combined with other terms. In DOM, a resharing step adds fresh randomness to these terms to make the sum statistically independent. Figure 3.5 shows the first-order

DOM multiplier [10]. The fresh random share r_0 is added to both of the cross-domain terms, meaning that the term is canceled when the shares are recombined. Furthermore, a register follows the addition of the random share to ensure r_0 has been added to the cross-domain terms before it is recombined with the inner-terms. This register secures the output against leakages through glitches when combining inner- and cross-domain terms.

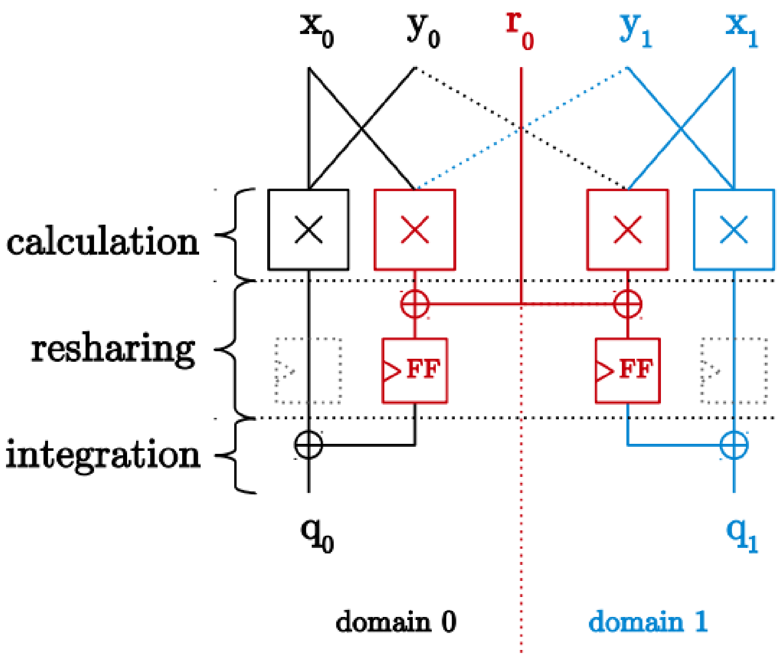

Figure 3.5: A generic depiction of a first-order DOM multiplier [10]

The first-order DOM multiplier can easily be extended for higher-orders. Figure 3.6 shows a second order-secure DOM multiplier. As for the first-order secure multiplier, fresh randomness is added to the cross-domain terms and followed by a register to prevent leakages due to glitching. Two fresh random shares are required for a second-order secure multiplier, which are canceled when recombining the output shares.

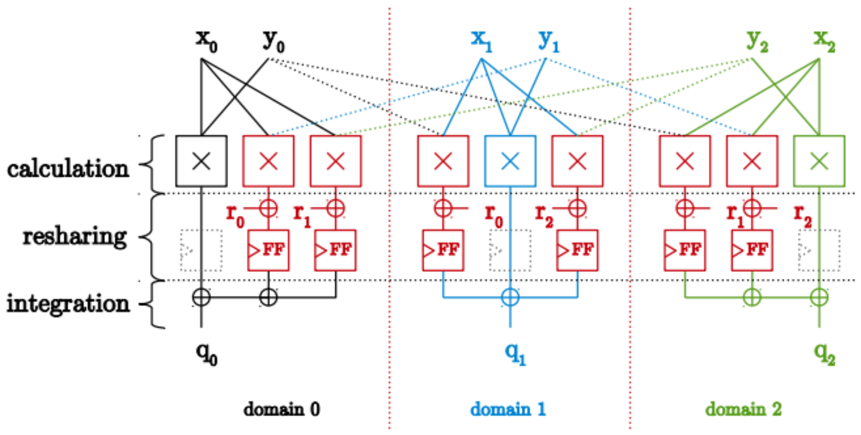

Figure 3.6: A second-order secure DOM multiplier [10]

Most of the sub-components of a composite field SBox are linear; only the multiplications and inversion in $GF(2^4)$, as shown in Figure 2.5 are non-linear. This subfield inversion can be further decomposed into three more non-linear multiplications in $GF(2^2)$, as shown in Figure A.2 in Appendix A. Thus, a hardened SBox using DOM utilize the DOM multipliers and some additional registers to protect against glitches.

Equation (3.9) shows the calculation of how much fresh randomness is required per SBox operation. There are three $GF(2^4)$ multiplications and three more $GF(2^2)$ multiplications that require four and two bits of fresh randomness per execution. The input byte must also be masked at the start by the additional shares. The bandwidth for fresh randomness required by the accelerator is the number of shares in the system subtracted by one and multiplied by 26. Hence, a two-share implementation requires 26 bits- and a three-share implementation would require 52 bits of fresh randomness for every SBox execution.

$$\begin{aligned}
&randombits = \\
&= (NrShares - 1) \times (DataSize + Mults_{GF(2^4)} \times 4 + Mults_{GF(2^2)} \times 2) \\
&= (NrShares - 1) \times (8 + 3 \times 4 + 3 \times 2) \\
&= (NrShares - 1) \times 26 \qquad (3.9)
\end{aligned}$$

Chapter 4

RISC-V

RISC-V is an open-source Instruction Set Architecture (ISA) [52]. The development of RISC-V has been ongoing since 2010 and has seen an increase in popularity since establishing the RISC-V foundation. More and more developers and companies are adopting the technology and contributing to its development, which helps strengthen its maturity.

4.1 RISC-V Modularity

One of the key strengths of RISC-V is its modularity, which enables the development of more specialized cores without an abundance of unused instructions. When developing a RISC-V core, one can start by selecting one of the five base instruction sets that contain the minimum number of instructions required for an operable core. Afterward, one can either extend this by adding custom instructions or using the official RISC-V Instruction Set Extensions (ISEs). There are five base instruction sets and 21 official ISEs for RISC-V, where some are still under development [53]. Table 4.1 lists some of the standard base and extension instruction sets.

Name	Description
Base Integer	
RV32I	32-bit Instruction Set
RV32E	32-bit with 16 registers Instruction Set for embedded
RV64I	64-bit Instruction Set
RV128I	128-bit Instruction Set
Standard Extension	
M	Integer Multiplication and Division
A	Atomic Instructions
F	Single-Precision Floating-Point
D	Double-Precision Floating-Point
C	Compressed Instructions
B	Bit Manipulation
K	Scalar Cryptography

Table 4.1: Some of the standard RISC-V base and extension instruction sets

4.1.1 Scalar-Cryptography Extension

The open-source scalar cryptography extension is one of the newly ratified extensions, which is available in [16]. This extension defines instructions and their behaviors for the two-block ciphers AES and ShangMi, the hash algorithm SHA2, a handful of bit manipulation instructions, and entropy that can be used to generate cryptographic secrets [54].

The technology used for security applications requires a thorough design process. Modern ciphers use large secret key sizes, making brute-force attacks infeasible with today's technology. Instead, researchers have turned their eyes to the hardware where they are implemented. As presented in Chapter 3, physical devices can leak information that is advantageous to an adversary. The design policies In the proposal for the scalar-cryptography extension for RISC-V, the developers present a set of policies that were used during development. They specify that the proposed instructions must not be timing-dependent to prevent timing side-channel attacks, but they do not apply any countermeasures against electromagnetic or power side channels. Instead, recommendations are given in the proposal where deemed relevant [54].

AES Instructions

The crypto extension defines four 32-bit instructions for Advanced Encryption Standard (AES): two for encryption (AES32ESI and AES32ESMI) and two for decryption (AES32DSI and AES32DSMI). There are two instructions per encryption/decryption in order to differentiate between the last round where MixColumns is not executed. The three or four letters at the end of the instruction names are abbreviations where *d* is decryption, *e* is encryption, *s* is sub-bytes, *m* is mix-columns, and *i* is immediate [55]. Figure 4.1 shows the format of an encrypt middle-round instruction, where bits [29:25] are used to define the type of instruction. Each instruction contains four variables: the immediate value *byte-select* (bs), source register addresses in *rs1* and *rs2*, and destination register address in *rd*. Additionally, the first seven bits declare the opcode for AES instructions. The expected behavior for this instruction is described in pseudo-code in Listing 2.

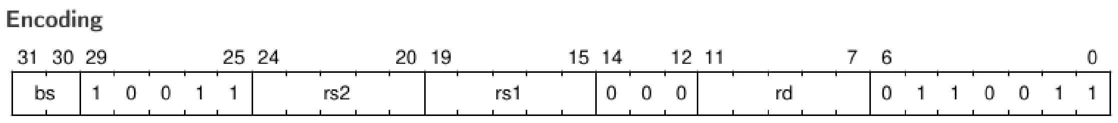

Figure 4.1: Format of AES32ESMI instruction which is a encrypt middle-round instruction [54]

```
function clause execute (AES32ESMI (bs,rs2,rs1,rd)) = {
let shamt : bits(5) = bs @ 0b000; /* shamt = bs*8 */
let si : bits(8) = (X(rs2)[31..0] >> shamt)[7..0]; /* SBox Input */
let so : bits(8) = aes_sbox_fwd(si);
let mixed : bits(32) = aes_mixcolumn_byte_fwd(so);
let result : bits(32) = X(rs1)[31..0] ^ rol32(mixed, unsigned(shamt));
X(rd) = EXTS(result); RETIRE_SUCCESS
}
```

Listing 2: Expected behavior of AES crypto instructions [54]

One of the criteria by National Institute of Standards and Technology (NIST) when selecting AES was that it had to be able to run on an 8-bit Central Processing Unit (CPU) [23]. The AES behavior specified by the Scalar-crypto extension uses a combination of 8-bit logic and 32-bit logic. Figure 4.2 gives a visual representation of the behavior described in Listing 2. The 16-byte state matrix is stored by columns in four 32-bit variables and input through source register *rs2*. One of the four bytes of each column is selected using the immediate *bs* variable. For each column of the new state matrix, the bytes in the previous state matrix must be input in the column order as if a ShiftRow operation has been executed, as was shown in Equation (2.10). The selected byte propagates through the Substitution-Box (SBox) and then the MixColumns, where it is multiplied by the vector [3, 1, 1, 2]. As shown in Chapter 2.3.4, each input byte is multiplied by a shifted version of the same vector. The last operation ShiftBytes shifts the four-byte output of MixColumns to match the corresponding shifted vector for the input byte. Four instructions are required for each column, totaling 16 instructions per round, which is one for each byte in the state matrix. The result of the three first instructions is input back through *rs1*, which accumulates a correct column result. Initially, the first input *rs1* contains the corresponding column of the round key.

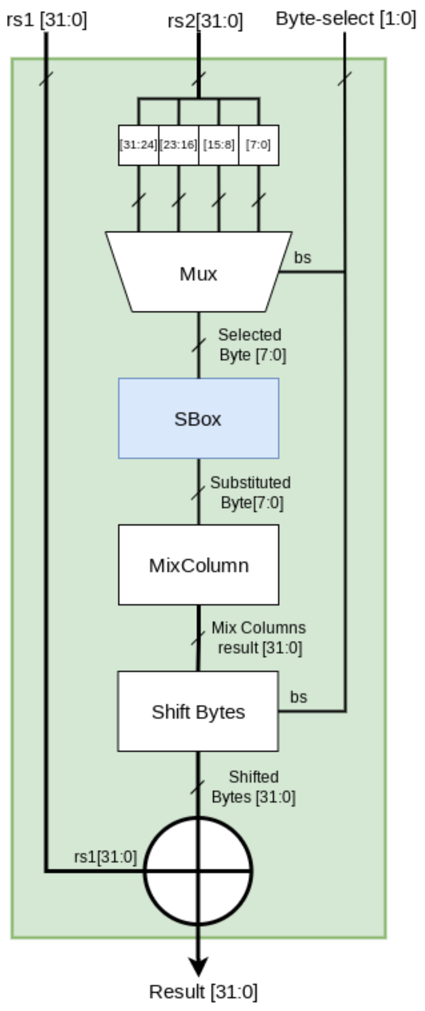

Figure 4.2: Microarchitecture of single byte AES implementation

Listing 3 shows how the instructions are called for one column in the input matrix; K0 contains 4-bytes of a round key column, while T_n contains the column values of the state. The diagonal is chosen from the matrix, which creates the row-shifting transformation of AES. These four instruction-calls modify the value of K0, which accumulates to the result of an entire column. Because of the commutative property of finite fields, the order of these instructions is not critical.

```
/* AES32ESMI rd, rs1, rs2, bs */
   AES32ESMI K0, K0, T0, 0
   AES32ESMI K0, K0, T1, 1
   AES32ESMI K0, K0, T2, 2
   AES32ESMI K0, K0, T3, 3
```

Listing 3: Instruction calls for a column in the state in a round

The scalar-crypto ISE includes an Hardware-Descriptive Language (HDL) implementation for the AES functional unit and an assembly application for encryption, decryption, and the key scheduler. The hardware implementation uses an unprotected Boyar SBox [32], which results in an implementation that may leak information through its side-channels.

4.2 Core-V Family

From the official RISC-V website, there are, at the time of writing this, 111 open-source cores that are available online, and more are under development [56]. Some of these cores belong to the Core-V family developed by the OpenHW group. The OpenHW group is a collaboration of individual developers and organizations who develop open-source Intellectual Property (IP), tools, cores, etc [14]. The Core-V family is a set of seven cores developed for various applications such as Internet-of-Things (IoT) and Unix Operating systems [15]. Silicon Labs have been a significant contributor to the RISC-V project, especially in the CVE4 subgroup. CVE4 is a group of small and efficient cores designed for IoT applications. The CVE4 cores started as a continuation of the RI5CY core from PULP with CV32E40P but were further extended with two new cores: CV32E40S for more secure applications [57] and CV32E40X for more compute-intensive applications [58].

4.2.1 CV32E40X Core

The CV32E40X is an open-source four-stage in-order 32-bit core with the four pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), execute (EX), and write-back (WB) [59]. All components used in the various pipeline stages can be seen in Figure 4.3. Instructions are fetched from memory to the core by the IF stage. The IF stage implements a prefetcher that fetches instructions even when the preceding stages are halted in an attempt to reduce the delay of fetching instructions from memory. The fetched instructions are decoded in the ID stage, where control signals are set, data is acquired from the instruction itself and from the register file, and prepared for the EX stage. What control signals are set and information acquired is dependent on the opcode and other control bits of the instructions. The execution stage implements four functional units for the various instructions: a load-store unit (LSU), control and status registers (CSR), Arithmetic Logic Unit (ALU), and multiplication/division unit (MUL/DIV). The result from EX is passed to the WB stage, where it is written back to the register file. The results from EX and WB can also be fed back to EX if the following instruction requires this result.

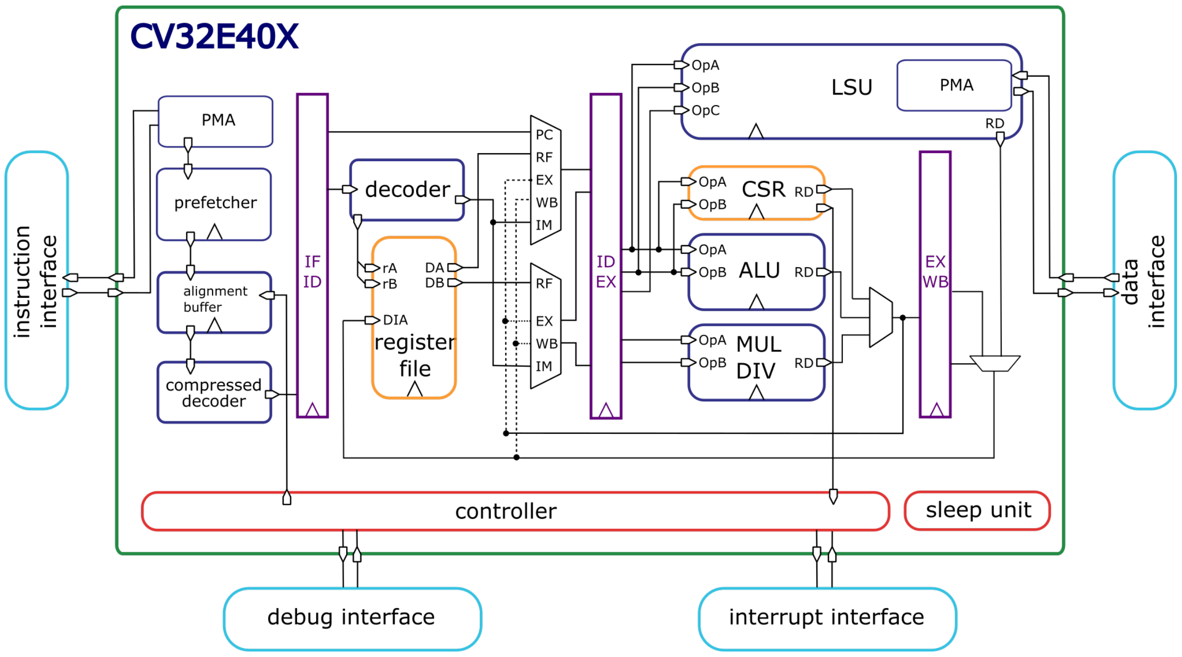

Figure 4.3: Block diagram for the CV32E40X core [59]

4.2.2 eXtension Interface

One of the more innovative technologies implemented in CV32E40X is the eXtension Interface (XIF), where coprocessors or accelerators can be connected to the core without modifying the design directly [60]. This enables developers to create accelerators in an isolated development and verification environment before connecting them to the more complex core pipeline. The XIF innovation is a consequence of the extendability of the RISC-V ISA. Figure 4.4 gives a visual representation of how the custom coprocessor is connected to the core pipeline.

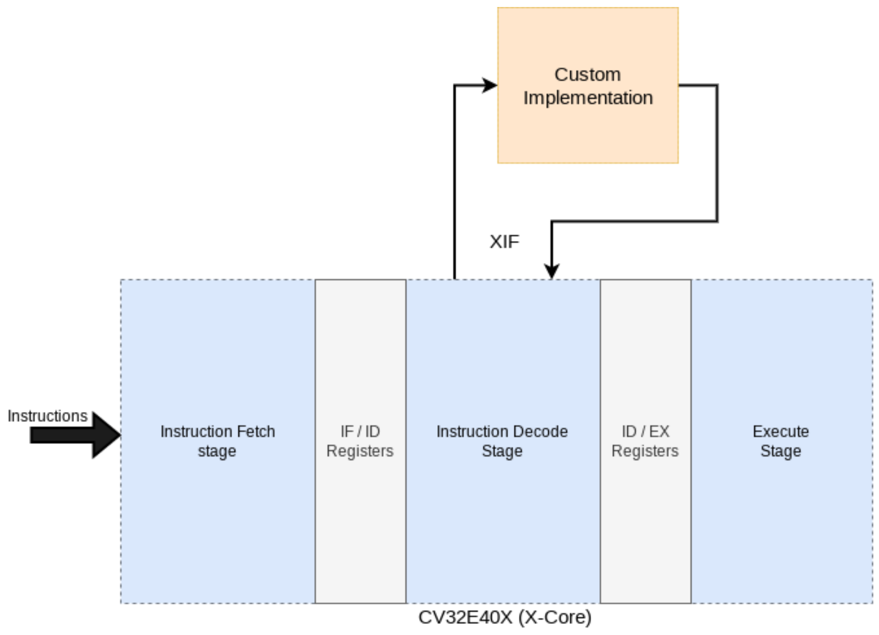

Figure 4.4: A coprocessor connected to a core using the XIF.

The XIF is split into six sub-interfaces: *compressed*, *issue*, *commit*, *memory*, *memory result*, and *result*. When the ID stage receives an instruction with an unknown opcode, the core attempts to offload it through the *issue interface*. The coprocessor must, in turn, reply with whether the instruction is accepted. The issue interface will pass on the instruction, register file values rs1, rs2, rs3 from addresses defined at instruction index [19:15], [24:20], [31:27], and an identifier for the instruction. If the instruction is compressed, it will first be offloaded to the coprocessor to be uncompressed, and then returned to the core, before it is posted back on the *issue interface* as a regular instruction.

Instructions that are offloaded to a coprocessor are speculative and there is a chance that it is invalid and must be flushed. For example, the ID stage must be flushed if the EX stage executes a branch instruction that is accepted. Since the instruction is already offloaded to a coprocessor at this point, the XIF must invalidate the instruction, which is done through the *commit interface*. Each instruction must be validated before the coprocessor writes back the results.

All load and store operations are requested through the *memory interface*. The coprocessor posts a request with the id of the instruction, the virtual address of the memory, the privilege level, the data to write (if it is a store), and information such as if it is the last memory transaction of the instruction and if it is speculative. The core can execute these operations as standard load/store instructions, and responds using the memory result interface with the data for a load operation and various information such as if it caused a bus error or if it caused a debug trigger.

When the coprocessor has finished its execution, and the commit interface has validated the instruction, the result can be written back through the *result interface*. The return packet includes the id of the instruction, the resulting data and its register file address, various status values, and exception codes.

Chapter 5

Design and implementation

5.1 Introduction

This chapter describes the practical design and implementation of this work. The task at hand is to implement a first-order Side-Channel Attack (SCA) secure Advanced Encryption Standard (AES) accelerator for the CV32E40X core [61] in the OpenHW Core-V family [15] using V1.0.1 of the scalar cryptography ISE [54]. CV32E40X implements a novel generic coprocessor interface eXtension Interface (XIF) [60], meaning that the accelerator can be developed without having to alter the internal Hardware-Descriptive Language (HDL) of the core. However, this also means a wrapper must be developed to connect it to the core. A thorough verification and simulation environment is available for the CV32E40X [18] where C or assembly applications are compiled and used as stimuli for simulation. An AES encryption and decryption application have also been developed using the implemented instructions to verify the behavior of the accelerator after a thorough development process. All the designs are available on GitHub and the structure of each repository is described in Appendix B.

Three main designs are developed in this work: the Substitution-Box (SBox), the complete accelerator, and the XIF wrapper. An overview of the complete system is shown in Figure 5.1. The CV32E40X pipeline is shown in the lower half of the figure in blue and gray, while the three levels of depth of the AES implementation are shown in varying shades of orange in the upper half.

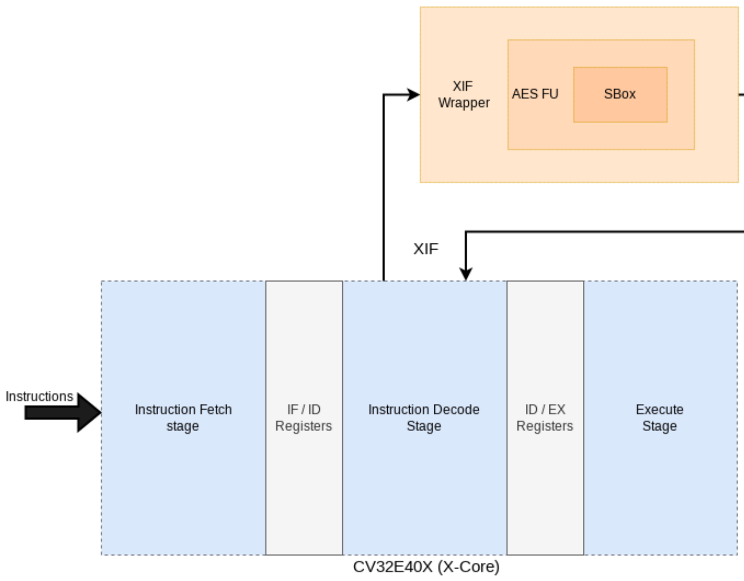

Figure 5.1: An overview of the CV32E40X core and how the AES accelerator will be connected to it.

5.1.1 Approach

The design process consists of three steps to facilitate the complex system's precise and thorough development: logic design using microarchitectures, behavioral verification through high-level models, and HDL implementation and verification in SystemVerilog.

The microarchitecture is a block diagram that illustrates all subcomponents and their connections. Large and complex designs consist of many components and connections that are hard to keep track of manually. A microarchitecture is a visual aid to keep track of the entire system, and it is a great tool for describing the design.

Debugging hardware implementations is a time-consuming and challenging task. The high-level code is a step to verify functional behavior before implementation in hardware. By verifying that functions produce the expected result, a developer can trust that the implementation should work. This model can also be used to acquire intermediate values for further debugging or verification.

After developing the design and verifying its behavior through the high-level model, the design is implemented in hardware using SystemVerilog and verified through the verification environment.

5.1.2 Chapter Structure

The rest of this chapter is structured as follows: firstly, the SBox design is presented and discussed in Section 5.2 since it is such a central component in the AES accelerator and one of the more complex components in the system. Afterward, the complete accelerator design is presented and discussed in Section 5.3, followed by the XIF wrapper in Section 5.4. In the end, a C application is presented, which uses the instructions in functions for key-scheduler, encryption, and decryption.

5.2 SBox Implementation

The principle of the Substitution-Box (SBox) was presented in section 2.3.2; in short, the SBox is a non-linear operation that consists of an inversion in $GF(2^8)$ and an affine transformation. Only the affine transformation is different from the encryption and decryption processes. This operation is typically implemented directly in hardware using the tower-field approach rather than storing a lookup table in Read-Only-Memory (ROM) for a more area-efficient design [29].

5.2.1 Masking Scheme

As described in Section 2.3.2, Canright's composite field inversion is area-efficient in hardware, but a straightforward implementation is not secure against SCAs. Since the non-linear component of the SBox, the $GF(2^8)$ inversion, is the most complex component and hardest to protect, it has received a lot of attention from researchers wanting to secure it. Some of the countermeasures against SCAs were presented in Section 3.3. One of the presented techniques is the *Domain Oriented Masking* scheme by Hannes Groß [10], which will be used in this work. This technique was chosen because of the thorough groundwork that was put into proving its hardness, in addition to its simplicity. It is similar to a standard threshold implementation scheme, but it requires less fresh randomness due to its composite field multiplier design, as shown in Figure 3.5. A generic version for higher-order implementations was also presented in [10], highlighting the simplicity of extending the scheme to higher-orders since the composite field multiplier the only non-linear component that require special attention in the protection scheme.

5.2.2 Microarchitecture

Implementing a Domain-Oriented Masking (DOM) scheme for the Canright inversion consists of two steps: share-splitting and protecting cross-domain terms. The complete AES accelerator will input two shares into the SBox, one for each domain. The linear operations of the SBox can be duplicated and operated on the inputs in parallel, while the non-linear operations must be modified. The first line in Equation 5.1 describes the composite field operations for the uppermost nibble of an inversion in $GF((2^4)^2)$, where X_h and X_l are the upper and lower nibble of the isomorphically transformed input, v is a constant from the irreducible polynomial, and Y_h is the higher nibble of the inverted result. To implement a two-share version of this operation, the input and output variables, X and Y, are split into (X_a and X_b) and (Y_a and Y_b). Equation 5.1 shows the implementation of share-splitting in the second line and the operations for the individual share domains in the following two lines. The lower nibble result equations are similar, except that the first multiplication uses the higher input nibbles instead of the lower nibbles.

$$Y_h = X_l[X_h X_l + (X_h + X_l)^2 v]^{-1}$$
$$\Downarrow$$
$$Y_{ah} + Y_{bh} = (X_{al} + X_{bl})[(X_{ah} + X_{bh})(X_{al} + X_{bl}) + (X_{al} + X_{ah})^2 v + (X_{bh} + X_{bl})^2 v]^{-1}$$
$$\Downarrow$$
$$Y_{ah} = X_{al}[(X_{ah} + X_{bh})(X_{al} + X_{bl}) + (X_{al} + X_{ah})^2 v + (X_{bh} + X_{bl})^2 v]^{-1}$$
$$Y_{bh} = X_{bl}[(X_{ah} + X_{bh})(X_{al} + X_{bl}) + (X_{al} + X_{ah})^2 v + (X_{bh} + X_{bl})^2 v]^{-1}$$

Equation 5.1: A two share split for the higher nibble of $GF(2^8)$ composite field inversion

The non-linear components of the inversion are the multipliers and the second inversion, which also consists of three subfield multiplications. The second step is to swap the multipliers with the DOM multipliers in both $GF(2^4)$ and in $GF(2^2)$. Figure 5.2 shows a shared inversion implementation using the DOM SBoxes. Parts of the figure are implemented following Equation 5.1. The higher nibble for each of the shares are output from the last multiplication, wherein the lower nibbles of the input are multiplied by the inverted result of the intermediate values. The $GF(2^4)$ inversion combines intermediate values of Share A and B. Both shares go through the square-and-scale stage where the sum of the higher and lower nibbles are squared and multiplied by a chosen parameter v. The result of this is added to the result of the multiplication and then inverted. A similar path is given for the lower output nibbles, but these are multiplied by the higher input nibbles.

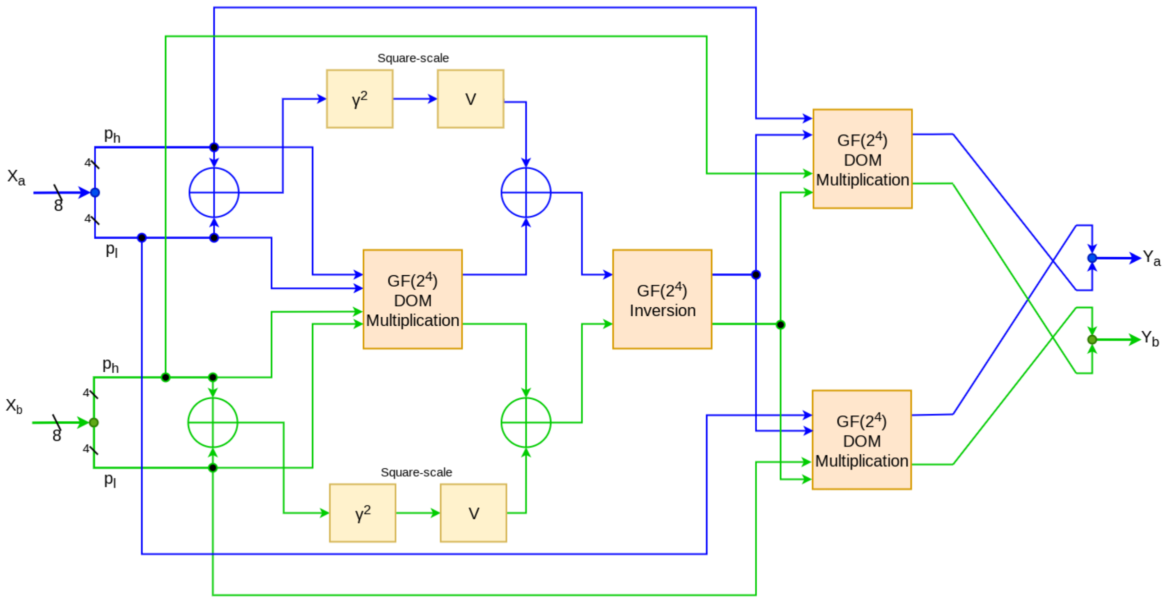

Figure 5.2: Non-pipelined version of DOM SBox. Differing colors represent the domains: blue for domain A and green for domain B.

The input to the SBox is originally in GF(2^8), and an isomorphic mapping must be used to convert to a value in the composite field GF($((2^2)^2)^2$) and reversed after finishing the inversion. The linear isomorphic transformation and its inverse used in this work were derived by Canright in [30] and are shown in matrix format and by their respective XOR chains in Equation 5.2a and Equation 5.2b.

$$\delta \times i = \begin{bmatrix} io_7 \\ io_6 \\ io_5 \\ io_4 \\ io_3 \\ io_2 \\ io_1 \\ io_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} i_7 \\ i_6 \\ i_5 \\ i_4 \\ i_3 \\ i_2 \\ i_1 \\ i_0 \end{bmatrix} = \begin{bmatrix} i_7 \oplus i_6 \oplus i_5 \oplus i_3 \oplus i_1 \oplus i_0 \\ i_6 \oplus i_5 \oplus i_4 \oplus i_0 \\ i_6 \oplus i_5 \oplus i_1 \oplus i_0 \\ i_7 \oplus i_6 \oplus i_5 \oplus i_0 \\ i_7 \oplus i_4 \oplus i_3 \oplus i_1 \oplus i_0 \\ i_0 \\ i_6 \oplus i_5 \oplus i_0 \\ i_6 \oplus i_3 \oplus i_2 \oplus i_1 \oplus i_0 \end{bmatrix}$$

(a)

$$(\delta \times i)^{-1} = \begin{bmatrix} io_7 \\ io_6 \\ io_5 \\ io_4 \\ io_3 \\ io_2 \\ io_1 \\ io_0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} i_7 \\ i_6 \\ i_5 \\ i_4 \\ i_3 \\ i_2 \\ i_1 \\ i_0 \end{bmatrix} = \begin{bmatrix} i_4 \oplus i_1 \\ i_7 \oplus i_6 \oplus i_5 \oplus i_3 \oplus i_1 \oplus i_0 \\ i_7 \oplus i_6 \oplus i_5 \oplus i_3 \oplus i_2 \oplus i_0 \\ i_6 \oplus i_1 \\ i_6 \oplus i_5 \oplus i_4 \oplus i_3 \oplus i_2 \oplus i_1 \\ i_7 \oplus i_5 \oplus i_4 \oplus i_1 \\ i_5 \oplus i_1 \\ i_2 \end{bmatrix}$$

(b)

Equation 5.2: Forward isomorphic transformation (a) and inverse isomorphic transformation (b)

The inversion that has been presented so far is only one of the two steps in the SBox operation. The second step is the affine transformation, which is different for encryption and decryption. It is a simple matrix multiplication and addition, as was shown in Section 2.3.2. Similar to the isomorphic transformation, this operations also converts into eight chains of XOR-gates as shown in Equation 5.3a and Equation 5.3b, where a_n are the bits of the input byte and b_n are the bits of the output.

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus 0 \\ a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus 1 \\ a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus 1 \\ a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \oplus 0 \\ a_7 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0 \oplus 0 \\ a_7 \oplus a_6 \oplus a_2 \oplus a_1 \oplus a_0 \oplus 0 \\ a_7 \oplus a_6 \oplus a_5 \oplus a_1 \oplus a_0 \oplus 1 \\ a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_0 \oplus 1 \end{bmatrix}$$

(a)

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a_6 \oplus a_4 \oplus a_1 \oplus 0 \\ a_5 \oplus a_3 \oplus a_0 \oplus 0 \\ a_7 \oplus a_4 \oplus a_2 \oplus 0 \\ a_6 \oplus a_3 \oplus a_1 \oplus 0 \\ a_5 \oplus a_2 \oplus a_0 \oplus 0 \\ a_7 \oplus a_4 \oplus a_1 \oplus 1 \\ a_6 \oplus a_3 \oplus a_0 \oplus 0 \\ a_7 \oplus a_5 \oplus a_2 \oplus 1 \end{bmatrix}$$

(b)

Equation 5.3: Forward affine transformation (a) and inverse affine transformation (b)

The complete SBox implementation is depicted in Figure 5.3. This pipelined version includes the isomorphic transformation before the inversion and the inverse transformation afterward; and the forward and inverse affine transformation used for encryption or decryption, respectively. One thing to remark in this implementation is how the vector addition of the affine transformation is only added to one of the shares. Since addition and subtraction is the same operation in extension fields with the prime characteristic *p=2*, as was presented in Section 2.2.3, adding the same element to an even number of shares would be canceled when recombining the shares. Thus, elements that should not be canceled out at the end must be added an odd number of times.

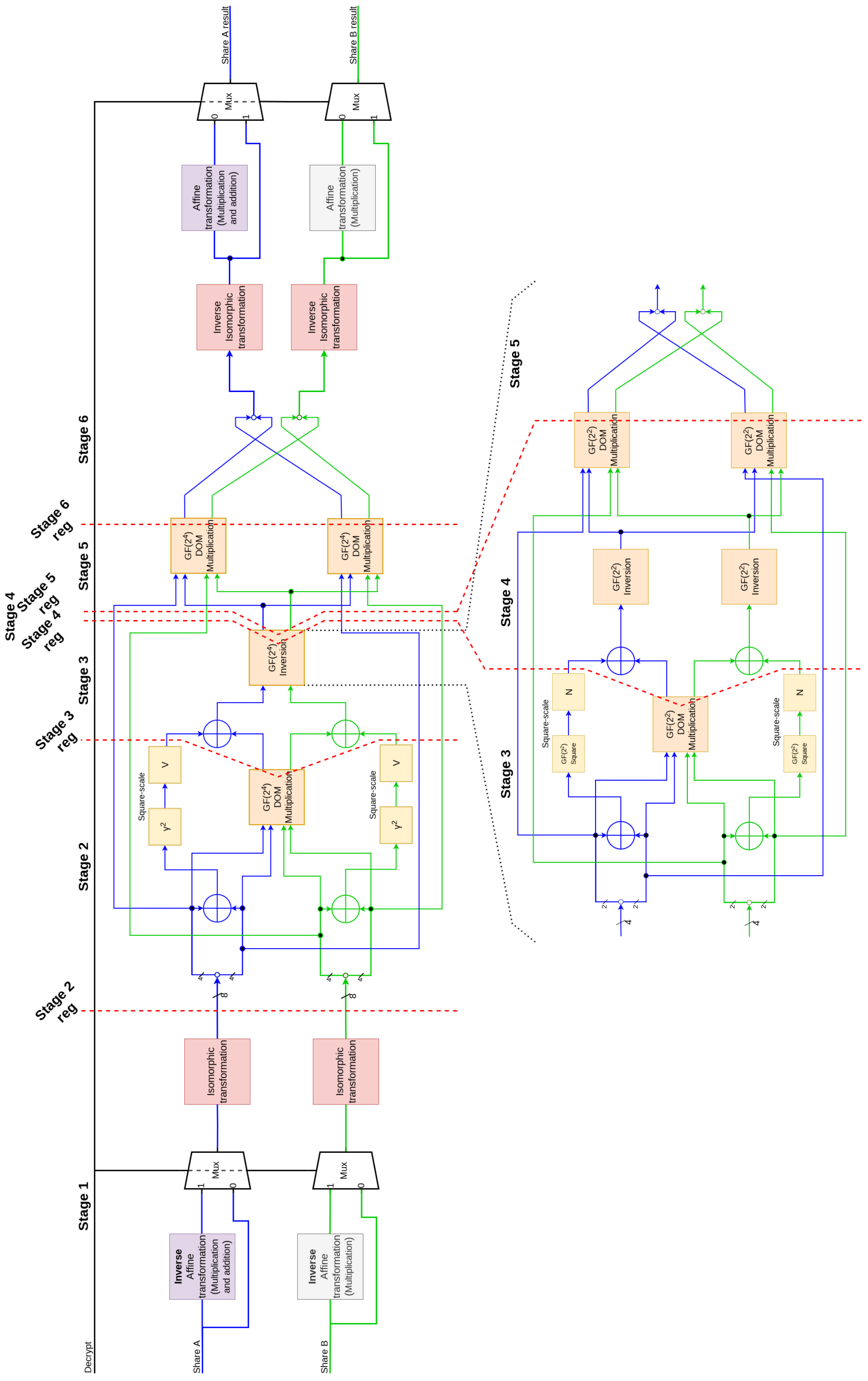
Decrypt
Share A
Share B
Inverse Affine transformation (Multiplication and addition)
Inverse Affine transformation (Multiplication)
Mux
Stage 1
Isomorphic transformation
Stage 2 reg
Stage 2
GF(2^4) DOM Multiplication
γ²
ν
Square-scale
Stage 3 reg
Stage 3
GF(2^4) Inversion
Stage 4 reg
Stage 4
Stage 5 reg
Stage 5
Stage 6 reg
Stage 6
Inverse Isomorphic transformation
Affine transformation (Multiplication and addition)
Affine transformation (Multiplication)
Share A result
Share B result
GF(2^2) Square
N
GF(2^2) DOM Multiplication
GF(2^2) Inversion

Figure 5.3: A complete implementation of a pipelined DOM SBox using two shares.

The pipeline stages are marked using red and stippled lines; five registers split the design into six pipeline stages. For data to pass from one stage to another, the following stage must be empty, or the stage ahead of that one must be ready to receive new data. Thus, data will not overwrite data when passing through the pipeline. The implementation follows a modified version of a valid/ready handshake protocol; when stage 2 is ready to receive, data is stored in the stage 2 registers, and new data can be input into stage 1. The valid signal on the input is also passed through the pipeline to enable or disable all operations of each stage and validate the output. Data can be overwritten in a pipeline stage if the valid signal is not high. An external ready signal is also input to the output stage, stage six, where data is passed out from stage six only when it is ready to receive.

5.2.3 Pipeline Discussion

The *five* pipeline registers splitting the SBox into six stages are required to provide security against glitching. Additionally, this reduces the critical path, which increases the maximum possible frequency.

The first stage implements the isomorphic transformations, where the length of the XOR chains used for this transformation, as seen in Equation 5.2a, have a variable number of gates per bit. Thus, signals combining data from various bits would arrive at different times in the second stage if the pipeline registers were not placed beforehand.

The DOM multiplier was described in Section 3.3.3. The stage three register is placed before the inner- and cross-domain products are combined to make sure that the fresh randomness is added to the cross-domain product first. Otherwise, if the inner- and cross-domain products were combined before the fresh randomness arrived, the glitch would have dependencies in both domains. The last three registers, stages four, five, and six, all follow the same logic as the stage three registers as they are used in the DOM multiplier.

5.2.4 Implementations

The DOM SBox has been implemented both in python as a high-level model [17] and in SystemVerilog [62]. The high-level model was first developed to verify the behavior and correctness of the implementation. After verifying the correctness of the high-level model, it was converted into SystemVerilog to be used in the AES accelerator.

5.3 AES Accelerator

The scalar crypto ISE has ratified a standard for developing AES accelerators for RISC-V [54]. In the publicly available repository, one can find detailed documentation of the standard, an example implementation of an AES accelerator, and complete encryption and decryption application written in assembly. The microarchitecture of this example implementation was presented in Figure 4.2, which has been used as a base to develop a first-order SCA secure AES accelerator. The following sections will elaborate on the chosen design, and in the end, there will be a discussion of the design choices that resulted in the current design.

5.3.1 Microarchitecture

The operational principle of the single-byte implementation was described in Section 4.1.1. Figure 5.4 illustrates a 2-share pipelined version of the AES accelerator. The design is split into four sections:

1 The original unprotected single-share AES flow in the green section

2 A duplicate flow for the second share in the orange section

3 Control and data signals in the purple section

4 Output stage in the yellow section

Simple linear operations such as addition, MixColumns, and byte shifting can be executed in parallel for both shares, so the green and orange are almost duplicates. The non-linear SBox is more complex and cannot be duplicated for each share. Instead, the DOM SBox presented in Section 5.2 has been used. There are two key differences between the green and orange flows: Share2 is added to Share1 at the start to mask the input data, and the key, input in rs1, is only added to one of the shares in the end. Which of the shares the key is added to is not significant, but the key must be added to only one of the shares not to cancel it out when the shares are recombined. A Random Number Generator (RNG) must feed the accelerator with 26 random bits for each instruction, where 8-bits are used to split the input into two shares, and the remaining bits are used in the SBox DOM multipliers.

The SBox must be pipelined to remove correlations that glitches can cause. Pipelining adds complexity to control the data propagation. The purple section of the figure shows the control signals used to input and output data without overwriting anything in any of the stages. Some signals are used after the SBox and must, therefore, follow along in the pipeline. This includes the byte-select and the instruction id used to identify the result at the accelerator's output. Other signals, such as decrypt and whether it is a middle round where MixColumns is executed or not, are also input but not depicted on the figure.

An initial pipeline register must be placed before the SBox to prevent leakage through glitches. If the selected input byte arrives at the XOR-gate of the second share, an unmasked version of the selected byte could propagate through the initial linear stage of the SBox. This glitch is prevented by placing a register after the share-splitting. Furthermore, the output of the byte-select mux is only enabled when there is a valid input, where both the random bits and input values must be valid. Again, this is to prevent operations from using unmasked values.

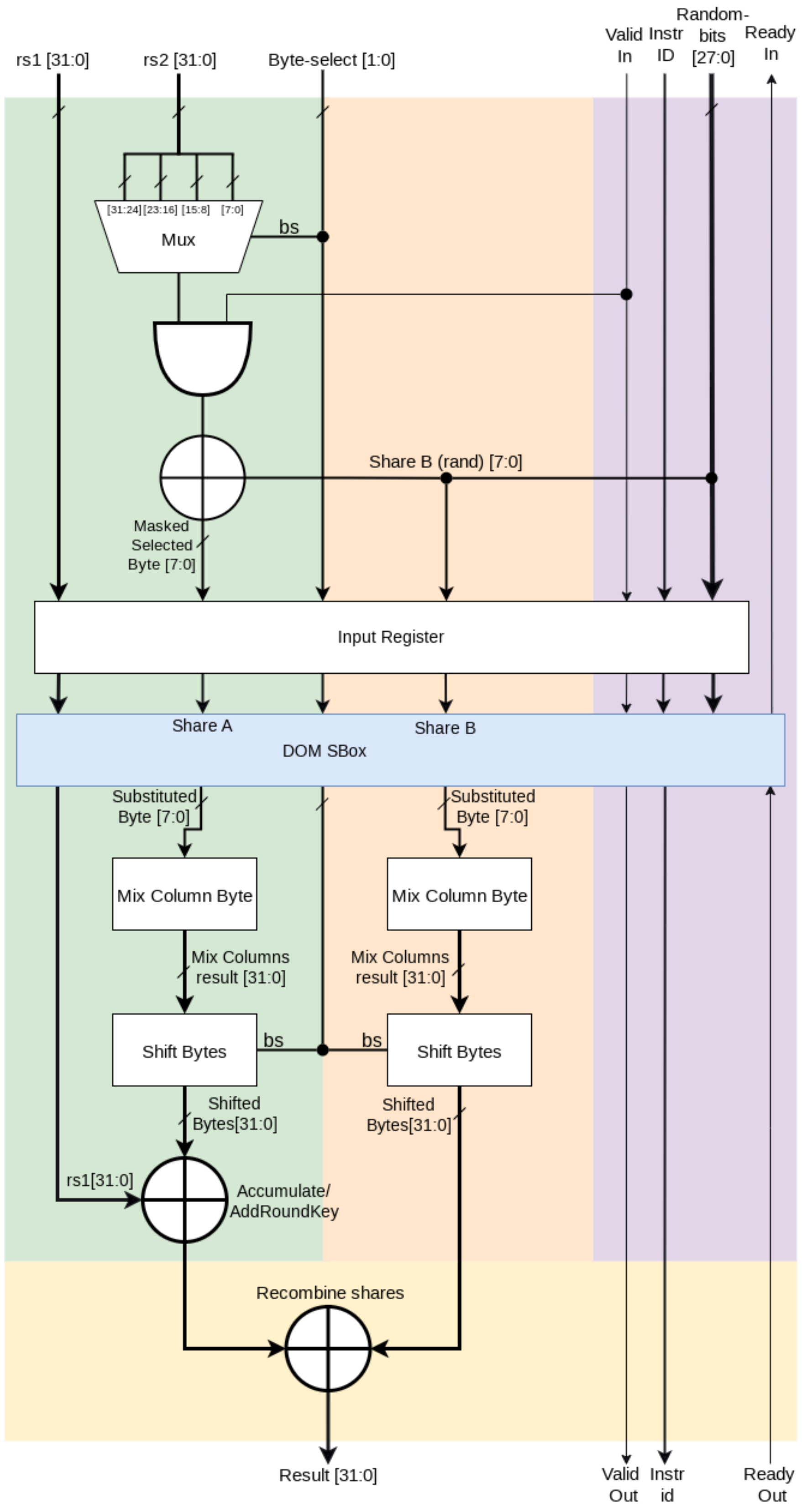

Figure 5.4: A 2-share AES accelerator implemented with a DOM SBox.

5.3.2 Design Discussion

An implementation utilizing share-splitting requires more data to create the shares and provide fresh randomness to the SBox. There are multiple approaches to supply this data depending on if the shares are created in hardware or software. During the design of the AES accelerator in this work, four different versions for supplying data were evaluated.

A core with an integrated RNG could fetch the required fresh randomness from this and create the shares in software. This means that the application is responsible for masking the input. All intermediate values for both shares are available and can be used as needed, but this also means that more data must be input to the accelerator through the instructions. The AES instructions defined by the Instruction Set Extension (ISE) standard only use two out of the three registers that XIF can forward. By moving and compressing the uppermost functional bits [29:25] to [14:12] and bs to [26:25], one could fit the third and last register to input more data in [31:27]. The instruction format would then be as illustrated in Figure 5.5 instead of the original in Figure 4.1.

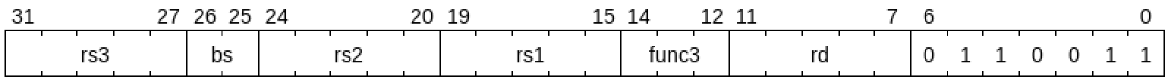

Figure 5.5: A modified version of the AES instructions considered to input more data into the accelerator.

Modifying the instructions would change the implementation to add custom instructions rather than following the official standard. The AES instructions are already implemented in the Embecosm compiler available at [63]. Changing the instructions would require a modified compiler that includes the custom instructions. There are plenty of resources online showing how to add custom instructions to compilers, such as [64]. When using standard ISEs, compilers and toolchains are typically maintained, while modifying a custom compiler adds the additional work of maintaining the compiler.

Instead of inputting the two shares using two registers, it would be possible to split one of the input registers into two 16-bit halves. The accelerator only needs to access one input byte at a time, freeing the other three bytes. Placing the shares in the correct register locations can be done using simple Arithmetic Logic Unit (ALU) instructions. A significant overhead would be incurred for such an application in order to move data into the correct locations.

A slightly modified wrapper could also be designed to collect data from multiple instructions before passing it on to the accelerator. This version would require a more complex XIF wrapper that can receive and prepare data from multiple instructions and then input this to the accelerator when it has received all it needs. Multiple instructions would be needed to insert the data, which would scale the application's size. Such an implementation would be easy to expand for higher-orders as another instruction could be used to input even more data.

The shares could also be created directly in hardware if the accelerator is connected to an RNG and can fetch the required random bits as they are needed. This means that the applications do not have access to any intermediate values during the masked execution, which could be considered a plus to minimize the information available for a hardened encryption accelerator. This approach was chosen for this work because it complies with the standard and requires no knowledge of non-standard instructions or behavior.

Table 5.1 summarizes the the four AES accelerator variations presented in the previous paragraphs. The implementation where share-splitting happens in hardware was chosen because it complies with the scalar-crypto ISE standard.

	Description	Complies with standard	Modified wrapper	Additional Work
1	Modify instructions to add a third register	No	Yes	Custom compiler
2	Split source register in 2x16-bits	No	Yes	Additional instructions for moving data into source register
3	Input data using multiple instructions	No	Yes	Additional instructions for inputting enough data
4	Share-splitting in hardware	Yes	No	RNG connected directly to accelerator

Table 5.1: Summary of evaluated approaches for supplying the accelerator with enough data

5.3.3 Implentations

The hardened AES accelerator has been implemented in both python as a high-level model [17] and in SystemVerilog [62]. The high-level model used the test vectors found in Federal Information Processing Standards (FIPS) [2] to verify its correctness. Once the correct behavior was achieved, the code could be manually converted into SystemVerilog. There are many similarities in the written code due to the high-level model being implemented as it would be in hardware. However, pipelining the design led to a more complex digital hardware design.

5.4 Extension Interface Wrapper

The AES accelerator described in the preceding chapters will be connected to the CV32E40X core using the extension interface. The interface works by attempting to offload an instruction if the Instruction Decode (ID) stage receives an unknown instruction. If successful, a coprocessor executes the instruction accordingly and returns the result. Further details can be found in Section 4.2.2.

A consequence of XIF being state-of-the-art is that there are few example implementations using it that are available online. In an attempt to contribute to the open-source development of RISC-V cores, one of the goals for the AES XIF wrapper is to develop a generic pipelined implementation, in the sense that it can handle n instructions simultaneously in its pipeline.

5.4.1 Microarchitecture

Figure 5.6 depicts the microarchitecture for the XIF wrapper. It consists of three sections: *accept* in green, *commit* in orange, and *output* in yellow. The first step of the wrapper is to check for a match in the instruction opcode; in other words, check if it is a known instruction for the accelerator. This check is done in the green *accept* section, where a valid signal is activated if there is a match, the XIF issue is valid, the accelerator is ready to receive, and the input source registers are valid. The accelerator must sample its inputs on the following clock cycle when the valid and ready signals are high. The wrapper is ready to receive a new input through the XIF if the accelerator is ready or there is no valid data on the input.

Since the core offloads instruction in its ID stage, the instructions are speculative. Hence, the core must verify each instruction before the implementation can write back its results. This validation is done through the commit interface whose logic is in the orange *commit* section. The instruction id and destination address *rd* for each accepted instruction are stored in an *accept First-in-First-Out (FIFO)* buffer. When the core attempts to validate or invalidate an instruction, the wrapper will compare the instruction id from the *accept FIFO* output to the incoming commit id. If there is a match, the *kill* information is added to the instruction id and rd address packet, and moved to the *commit FIFO*. When the accelerator has a valid output, the coupled output id is compared to the *commit FIFO* output id. The interface output result becomes valid if there is an id match, the output of the accelerator is valid, and the instruction was not *killed* by the core. The output stage of the wrapper is ready to receive new data if the XIF result ready signal is activated or if there is no valid output from the accelerator.

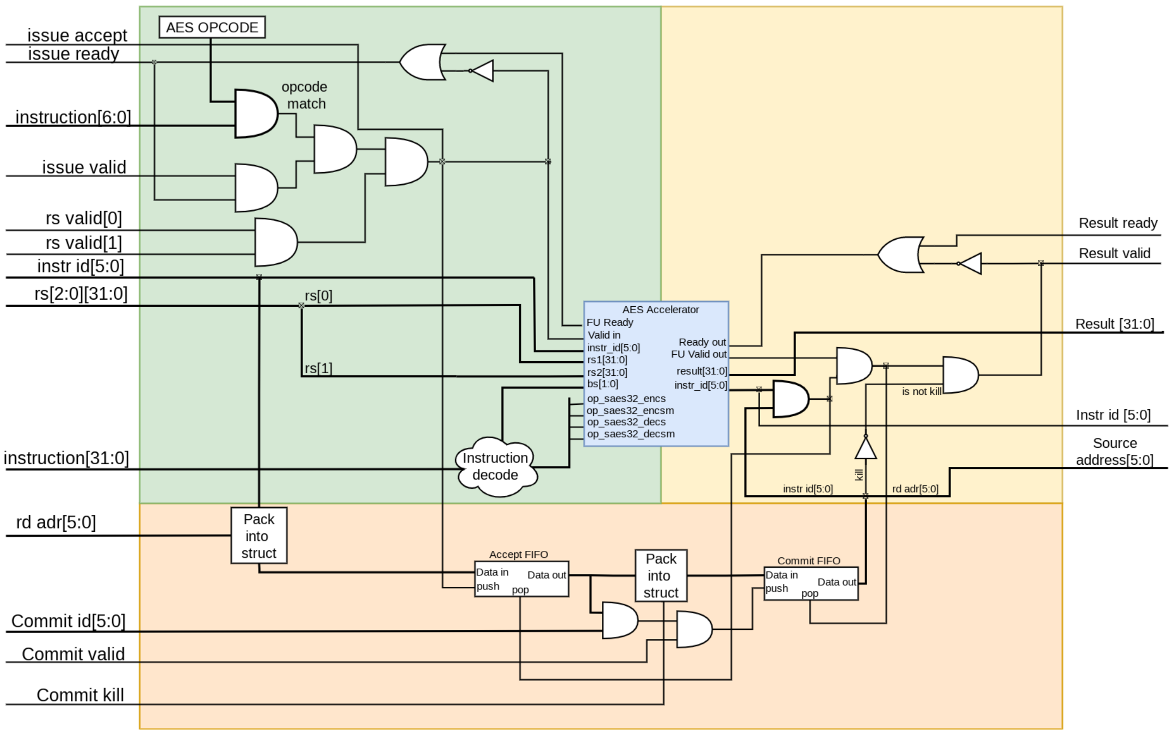

Figure 5.6: eXtension interface wrapper with its three sections: accept in green, commit in orange, and output in yellow.

If a purely combinatoric accelerator were implemented using this wrapper, a set of registers would have to be placed before the accelerator to store the input data, or the accelerator itself would have to implement input registers. Furthermore, the FIFOs could be replaced by some additional logic to slightly reduce the area.

The number of entries in, or depth of, the FIFOs depends on the design. The *commit FIFO* should have a depth according to the max number of instructions that can be executed sequentially at the same time in the accelerator. For the AES wrapper, the *commit FIFO* has a depth of four, which will be discussed in Section 6.1.3.

5.5 AES Application

The design in this work has, up to this point, been verified using the high-level model developed for each stage. Nevertheless, the correctness of the hardware implementation must also be verified. The OpenHW group has developed a large verification environment for each of the cores in the Core-V family [18]. This environment uses a state-of-the-art compiler from Embecosm [63] developed for the Core-V family, where instructions for all relevant ISEs are embedded. Instructions from the RISC-V scalar-crypto ISE are implemented and can be enabled using the flags *"zkne1p00"* and *"zknd1p00"* for the *"march"* compile flag.

The scalar-crypto ISE provides an example assembly application for their implementation. For this work, a new application has been developed in C using inline assembly instead to provide a more comprehensible and readable application. Inline assembly enables the use of assembly instructions directly in C. Listing 4 shows an example with an AES encryption instruction with inline assembly. The volatile keyword at the start notifies the compiler to not optimize this call, *bs* is the byte-select constant, while %0, %1, and %2 are the input and output variables or registers.

```
asm volatile("aes32esi %0, %1, %2, bs": "=r"(out): "0"(in0), "r"(in1));
```

Listing 4: Example of an AES instruction call in c using inline assembly

There are four main functions included in the AES application, which are further elaborated in the following sections.

1 KeyScheduler

2 Encrypt

3 Decrypt

4 Equivalent Decryption RoundKeys

A complete application implementing all these functions and an example of encryption and decryption can be found in [65]. This application can be used for both the protected AES implementation developed in this work and for the unprotected example available in the scalar-crypto ISE [54]

5.5.1 KeyScheduler

The KeyScheduler algorithm was described in Section 2.3.5. For each N-word, the KeyScheduler must do a word rotation, a Byte Substitution for the four bytes in the word, and add a round constant to the result. Furthermore, if the input round key has more than six words, each fourth block must run through the Byte Substitution.

Adding a round constant and rotating the bytes in a word are simple ALU operations. Listing 5 shows the entire process executed for each N-th word, where ROTATE_32_LEFT is a macro function that shifts the bytes in a 32-bit word to the right by one and inserts the byte that falls of as the most significant byte; rcon is an array of round constants calculated following Equation 2.13; and N is the number of 32-bit words in the secret key. Using the encryption instruction AES32ESI, only the SBox is executed. The listing shows how four instructions are used to substitute each byte in a word, where each individual byte is selected from the variable *word* using byte-select 0-3. The result of each instruction accumulates in the round_keys variable, which is why it is input through source register *rs1* and destination register *rd*.

```
round_keys[i] = 0;
uint32_t prev_key_word_rotated = ROTATE_32_LEFT(round_keys[i - 1]);

asm volatile("aes32esi %0, %1, %2, 0":"=r"(round_keys[i]):
↪  "0"(round_keys[i]), "r"(prev_key_word_rotated));
asm volatile("aes32esi %0, %1, %2, 1":"=r"(round_keys[i]):
↪  "0"(round_keys[i]), "r"(prev_key_word_rotated));
asm volatile("aes32esi %0, %1, %2, 2":"=r"(round_keys[i]):
↪  "0"(round_keys[i]), "r"(prev_key_word_rotated));
asm volatile("aes32esi %0, %1, %2, 3":"=r"(round_keys[i]):
↪  "0"(round_keys[i]), "r"(prev_key_word_rotated));

round_keys[i] = round_keys[i] ^ (rcon[i/N - 1] << 24);
round_keys[i] = round_keys[i] ^  round_keys[i - N];
```

Listing 5: Procedure executed for each N-word in the KeyScheduler algorithm.

5.5.2 Encryption

Encryption, as described in Chapter 2.3, consists of three main steps: a lone AddRoundKey at the start, Nr-1 rounds with MixColumns, and the last round without MixColumns. The initial AddRoundKey is a simple XOR operation that can be performed by the ALU, while the other two steps use the instructions AES32ESMI and AES32ESI, respectively. The procedure is the same for a middle round and the last round, but the instructions change. Listing 6 shows a middle-round encryption implemented in C. As observable in the listing, 16 instructions must be used for each round, reflecting the 16-bytes or 128-bits block size. The instructions execute the byte substitution, mix columns, and AddRoundKey, but the shift rows operation is done by selecting bytes from the words as if they were rotated. The last instruction for each column writes the column result back to the state matrix.

```
uint32_t key_tmp[4] = {round_keys[i*4+0], round_keys[i*4+1],
↪  round_keys[i*4+2], round_keys[i*4+3]};
uint32_t cipher_tmp[4] = {cipher[0], cipher[1], cipher[2], cipher[3]};

asm volatile("aes32esmi %0, %1, %2, 0": "=r"(key_tmp[0]) : "0"(key_tmp[0]),
↪  "r"(cipher_tmp[0]));
asm volatile("aes32esmi %0, %1, %2, 1": "=r"(key_tmp[0]) : "0"(key_tmp[0]),
↪  "r"(cipher_tmp[1]));
asm volatile("aes32esmi %0, %1, %2, 2": "=r"(key_tmp[0]) : "0"(key_tmp[0]),
↪  "r"(cipher_tmp[2]));
asm volatile("aes32esmi %0, %1, %2, 3": "=r"(cipher[0] ) : "0"(key_tmp[0]),
↪  "r"(cipher_tmp[3]));

asm volatile("aes32esmi %0, %1, %2, 0": "=r"(key_tmp[1]) : "0"(key_tmp[1]),
↪  "r"(cipher_tmp[1]));
asm volatile("aes32esmi %0, %1, %2, 1": "=r"(key_tmp[1]) : "0"(key_tmp[1]),
↪  "r"(cipher_tmp[2]));
asm volatile("aes32esmi %0, %1, %2, 2": "=r"(key_tmp[1]) : "0"(key_tmp[1]),
↪  "r"(cipher_tmp[3]));
asm volatile("aes32esmi %0, %1, %2, 3": "=r"(cipher[1] ) : "0"(key_tmp[1]),
↪  "r"(cipher_tmp[0]));

asm volatile("aes32esmi %0, %1, %2, 0": "=r"(key_tmp[2]) : "0"(key_tmp[2]),
↪  "r"(cipher_tmp[2]));
asm volatile("aes32esmi %0, %1, %2, 1": "=r"(key_tmp[2]) : "0"(key_tmp[2]),
↪  "r"(cipher_tmp[3]));
asm volatile("aes32esmi %0, %1, %2, 2": "=r"(key_tmp[2]) : "0"(key_tmp[2]),
↪  "r"(cipher_tmp[0]));
asm volatile("aes32esmi %0, %1, %2, 3": "=r"(cipher[2] ) : "0"(key_tmp[2]),
↪  "r"(cipher_tmp[1]));

asm volatile("aes32esmi %0, %1, %2, 0": "=r"(key_tmp[3]) : "0"(key_tmp[3]),
↪  "r"(cipher_tmp[3]));
asm volatile("aes32esmi %0, %1, %2, 1": "=r"(key_tmp[3]) : "0"(key_tmp[3]),
↪  "r"(cipher_tmp[0]));
asm volatile("aes32esmi %0, %1, %2, 2": "=r"(key_tmp[3]) : "0"(key_tmp[3]),
↪  "r"(cipher_tmp[1]));
asm volatile("aes32esmi %0, %1, %2, 3": "=r"(cipher[3] ) : "0"(key_tmp[3]),
↪  "r"(cipher_tmp[2]));
```

Listing 6: A middle-round of AES encryption using the scalar-crypto instructions

Figure 5.7 depicts an example signal flow from when the core offloads and instruction through XIF to when the wrapper writes the result back to the core. The XIF couples an id to the instruction before offloading it on the issue interface and expects to see the corresponding id when the wrapper writes the result back if the instruction has been committed. Once the wrapper accepts the instruction, data from the source registers are input to the accelerator, and data from the instruction, such as bs and instruction type, are decoded and also sent to the accelerator. After the data has propagated through the pipeline, the wrapper checks if the id has been committed and, if so, validates the data on the XIF result interface. New data can be passed to the output stage once the core has accepted the output from the result interface.

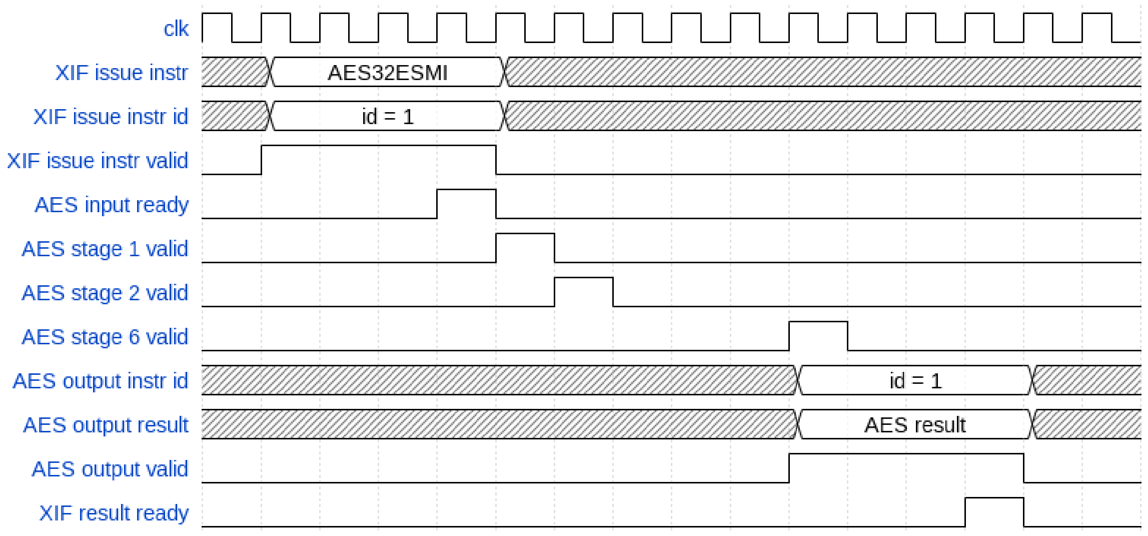

Figure 5.7: Signal flow from when an instruction is offloaded to when it is written back to the core.

5.5.3 Decryption

Decryption is identical to the encryption algorithm, except that it uses the AES32DSMI and AES32DSI instructions, and the round keys are used in the reverse order, starting with the last key. Furthermore, since the hardware implementation uses the same order of operations for encryption and decryption, the equivalent decryption procedure must be used. This means that the round keys must go through the inverse MixColumns before they are used in decryption as described in 2.3.6.

5.5.4 Equivalent Decryption Round Keys

Using the round keys in the decryption algorithm must first go through the inverse MixColumns operation. There is no direct way to perform the inverse MixColums operations alone, but a combination of instructions can be used. Using an AES32ESI and then an AES32DSMI, the byte substitution is canceled, and only the inverse MixColumns is left. This is not done for the first and last four key words. Listing 7 shows the instructions used to create the decryption round keys.

```
uint32_t tmp = 0;
asm volatile("aes32esi %0, %1, %2, 0": "=r"(tmp) : "0"(tmp),
↪  "r"(key_words[i]));
asm volatile("aes32esi %0, %1, %2, 1": "=r"(tmp) : "0"(tmp),
↪  "r"(key_words[i]));
asm volatile("aes32esi %0, %1, %2, 2": "=r"(tmp) : "0"(tmp),
↪  "r"(key_words[i]));
asm volatile("aes32esi %0, %1, %2, 3": "=r"(tmp) : "0"(tmp),
↪  "r"(key_words[i]));

uint32_t tmp2 = 0;
asm volatile("aes32dsmi %0, %1, %2, 0": "=r"(tmp2) : "0"(tmp2), "r"(tmp));
asm volatile("aes32dsmi %0, %1, %2, 1": "=r"(tmp2) : "0"(tmp2), "r"(tmp));
asm volatile("aes32dsmi %0, %1, %2, 2": "=r"(tmp2) : "0"(tmp2), "r"(tmp));
asm volatile("aes32dsmi %0, %1, %2, 3": "=r"(tmp2) : "0"(tmp2), "r"(tmp));

decryption_key_words[i] = tmp2;
```

Listing 7: Algorithm that creates decryption round-keys for the equivalent procedure

Chapter 6

Results and Discussion

The implementations themselves are the main contributions of this work. In this chapter, the size and performance of the various designs are compared to discuss the cost of security. Furthermore, the security of the hardened implementation is evaluated and discussed.

6.1 The Cost of Security

Area and throughput are typically used to quantize a digital design. The more components, the larger the area and the higher the cost. An area comparison before and after implementing the countermeasure will clearly present the cost of increased security. The area is defined using the unit Gate Equivalent (GE), which is the total area over the area of a 2-input NAND gate. To give basis for others when comparing designs, the area of an XOR-gate is 2.33 GE, and an AND-gate is 1.33 GE for the library used in this work.

The first-order Side-Channel Attack (SCA) secure Advanced Encryption Standard (AES) accelerator will be compared to the baseline design that was developed for the scalar-crypto Instruction Set Extension (ISE) [16] to present the area and throughput overhead caused by the additional security.

6.1.1 Area of Implementations

Table 6.1 decomposes the total area for the baseline on the left and Domain-Oriented Masking (DOM) pipelined on the right into their respective subcomponents, giving a clear picture of what the more costly components are. The 2-share pipelined design is significantly larger than the baseline, wherein the Substitution-Box (SBox) has the most significant contribution. Within the SBox, pipeline registers are the most significant contribution, taking up 65% of the area. The SBox in the scalar-crypto example implementation also significantly contributes to their implementation area, but the overall area is small since they utilized the Boyar SBox implementation.

There is also a significant difference in the area of the eXtension Interface (XIF) wrappers. The baseline version requires input registers before the accelerator since it is purely combinational, while the input registers are moved into the accelerator for the protected implementation. As will be discussed in Section 6.1.3, the accelerator needs to hold at most four instructions at a time. Hence, the second First-in-First-Out (FIFO) must have a depth of four instructions. The FIFOs contribute to most of the area in the pipelined wrapper.

Baseline

Component	**Area [kGE]**	**% of total**
XIF wrapper	814.9	34.7%
Accelerator with SBox	1535.6	65.3%
MixColumns	252.2	10.7%
SBox	720.82	30.7%
Total	2350.2	100%

DOM pipelined

Component	**Area [kGE]**	**% of total**
XIF Wrapper	1469.9	21.4%
FIFOs (2+4 depth)	1230.91	18.4%
Accelerator With SBox	5254.6	78.6%
MixColumns	504.4	8.9%
SBox	3763.7	56.3%
Total	6687.7	100%

Table 6.1: Table decomposing the areas of the baseline and DOM pipelined implementations into their respective subcomponents

The EX stage of a core typically implements some of the more complex components. The areas of the Arithmetic Logic Unit (ALU) and multiplication unit from the CV32E40X core are given in Table 6.2 with the area of the pipelined DOM implementation. Comparing the core components area and the protected AES accelerator area, it is clear that the hardened AES design is comparable to the other EX-stage components. On the other hand, the AES accelerator can only execute four specific instructions, while the generic ALU can execute more than 40 instructions. Hence, the cost per instruction is on the higher side for this AES implementation.

Component Name	**Area [kGE]]**
ALU	6772
MULT/DIV	4859
Protected AES	6688

Table 6.2: Area of CV32E40X core EX stage components in comparison to the area of the AES accelerator

6.1.2 Critical Path

The protected implementation requires the use of registers in order to hinder leakage caused by glitches. The additional pipeline registers significantly increase the implementation area; furthermore, the maximum frequency of the design can increase if the critical path has been shortened. The highest clock frequency a design can use is inversely proportional to the delay through the critical path. Table 6.3 summarizes the critical path and the maximum frequency for the baseline and pipelined implementations. The critical path delay was given through the synthesis of the implementations. The baseline and the pipelined implementations have a short critical path and, therefore, a high max frequency.

Version	**Critical Path [ns]**	**Frequency [MHz]**
Baseline	2.0	500
DOM Pipelined	1.5	667

Table 6.3: Critical path and maximum frequency for baseline and DOM AES accelerator

6.1.3 Throughput Performance

The pipeline increases the maximum allowed frequency due to decreasing the critical path, but it will also significantly impact the data throughput. Similar to the pipelined flow of a core, the AES pipeline is split into stages that can execute their operations individually. Thus, if the core and application are optimized and able to take full advantage of the pipelined structure, the impact on throughput in the protected implementation should, in theory, be insignificant.

Table 6.4 present the generic equations for approximating the number of clock cycles required for a full encryption or decryption procedure. These approximations are based on the assumption that all instructions are input sequentially, with no other instructions in-between. In a more practical example, there would be additional instructions for moving data and controlling the flow. The throughput of the pipelined implementation is split into the best and worst approximations. The best is if the pipelined is fully taken advantage of, and the worst is if instructions must propagate fully through the pipeline before the following instruction can start.

	Generic Equation	**AES-128**	**AES-256**
Baseline	$16 \times Rounds$	160	224
Pipelined$_{best}$	$baseline + 2 \times pipeline\_stages$	172	236
Pipelined$_{worst}$	$baseline \times pipeline\_stages$	960	1344

Table 6.4: Throughput comparison of AES accelerator implementations in clock-cycles, where the pipelined implementation has 6-stages

The implemented AES design executes instructions sequentially like a standard pipeline. However, the functions written for the application in Section 5.5 cannot utilize the pipeline fully as that would cause a Write-Wfter-Read (WAR) hazard [3]. A WAR hazard is when one instruction loads a value before it is updated by an instruction executed before it. Four instructions are executed for each column in the AES encryption and decryption functions, whose results are accumulated in a single 32-bit register. The first instruction must write back its result before the next instruction can get the correct input. Some of the functions could be optimized to enable one instruction call from each column to execute sequentially, i.e., increasing the throughput of the worst result by a factor of four. In summary, a straightforward AES application cannot take full advantage of the pipeline, but it is possible to optimize the instructions to improve the worst-case throughput presented in Table 6.4, resulting in throughput between the best and the worst.

6.1.4 Additional Costs

One of the strengths of the design used in this work is that the behavior of the accelerator follows the description of the Scalar-Crypto ISE. The area and performance are significantly impacted, but the original application can be used without modification. Thus, no additional knowledge is required, and the application size stays the same. The accelerator must be connected directly to an Random Number Generator (RNG) to achieve the desired behavior, which adds to the cost. Since RNG is a critical component for masking, its design must be thorough and secure. The RNG is one of the topics in National Institute of Standards and Technology (NIST)'s roadmap for standardizing threshold implementations for cryptography [19]

6.2 Security Evaluation of the Implementation

The security of the accelerator in this work has not yet been verified by formal verification nor by performing a physical SCA. The implementations have been based on the thorough foundational work from various state-of-the-art countermeasures, creating a reasonable basis. However, other leakages may have been overlooked during development, and the security of the design must be formally verified and physically tested against SCAs to verify that it is sound. Furthermore, only SCAs have been considered in this work. Injecting faults into the design to induce faulty behavior can lead to further leakages that must be considered. This chapter will attempt to elaborate on some security issues that should be discussed and considered beforehand.

6.2.1 8-Bit Implementation

One of the criteria from NIST when selecting AES was that the cipher should be able to run on an 8-bit Central Processing Unit (CPU) [13]. The scalar-crypto ISE takes advantage of this behavior in their example implementation. In such an implementation, the instructions are executed sequentially, while operations are typically executed in parallel for larger accelerators that operate on full state columns. Performing SCAs on implementations doing parallel operations is more difficult than sequential ones since the power consumption will overlap information from all the parallel operations [44]. The baseline implementation used in this work is both sequential and unprotected, but the protected implementation uses the share-splitting scheme, which increases the number of operations executed in parallel and, thus, adding additional noise by overlapping operations.

6.2.2 Expanding the Perspective

Taking a step back and changing the perspective to a CPU that implements the AES accelerator, opens for various security questions and considerations. The input to the accelerator, the round keys, and state-matrix columns, are stored in external memory or in registers in the CPU. The accelerator masks the input value with fresh randomness fetched from an RNG. Hence, the inputs are untouched and contain secret values. The debug interface is used during development to debug and verify the hardware. It implements powerful debug features such as stepping through instructions, reading registers, and downloading the firmware. This interface is typically disabled once the debugging phase of development has finished. Suppose the interface is disabled during startup by setting a bit in the registers. In that case, fault-injection attacks such as clock- or voltage glitching can skip instructions that disable the interface [66]. Acquiring access to the debug interface would circumvent the security of the cipher since its secret values could be read in plaintext.

Instead of hardcoding critical values such as the secret key directly into the firmware, a secure and inaccessible memory could be used to store critical values. Some Internet-of-Things (IoT) systems integrating a high level of security already do this, such as the local company Disruptive Technologies [67]. Storing critical values in a secure and inaccessible memory would make these safe initially. However, the critical values are still stored in the core registers when using the scalar-crypto AES algorithm. A complete AES accelerator could be implemented in order to remove this threat by directly accessing secure memory and being input blocks from the core. Implementing a complete accelerator would disregard the standard behavior but could potentially be more secure with a significant increase in area. Table 6.5 presents the area of three complete AES-128 accelerators. The areas of the first two are significantly larger than the accelerator in this work, but the area of the first-order secure DOM accelerator is comparable.

Complete AES-128 Accelerator	**Area [kGE]**
Compact Threshold implementation [68]	11.1
Nimble Threshold Implementation [69]	8.1
First-order SCA secure DOM [10]	6.0

Table 6.5: The area of complete AES-128 accelerators from research papers and theses

Attempting to get access to the debug interface is often the place one starts for fault-injection attacks. Other fault-injections techniques attempt to corrupt data that is used during encryption and compare the corrupted result with the unmodified result. Differences between a corrupted and non-corrupted result could leak information such as the secret key [70]. Side-channels could also be advertently integrated by an adversary during chip-production as a trojan [71]. Hardware trojans are defined as malicious modifications by untrusted third parties and may open back-door access to devices [72]. Depending on the security level of the applications, these attack vectors should be considered during development.

6.2.3 Extending Design for Higher-Order Protection

As discussed in Section 5.2.1, the DOM scheme was selected because of its generic design that can easily be extended for protection against higher-orders. Hence, extending the accelerator in this work to a higher-order is not a monumental task. A DOM implementation is secure against *d-th* order attacks when using $d + 1$ shares, and so, three shares are required to make the accelerator secure against second-order SCAs. Another duplicate flow similar to the orange one in Figure 5.4 would be created for the third share, which would also be added to the first share. The SBox would also have another similar flow to either of the domains in Figure 5.3, but the multiplier must be swapped for the second-order multiplier shown in Figure 3.6. These changes would make the implementation second-order secure with a significant area increase. Additionally, the number of random bits required for each instruction would increase as described in Section 3.3.3. A three-share implementation would require 52 bits of fresh randomness for each instruction call.

The required security of an implementation is heavily dependent on the application. First-order attacks have freely available resources online that reduce cost of getting started with SCAs. Higher-order attacks are significantly more complex, but with the development of moderns techniques and employment of machine learning algorithms, the complexity of higher-order attacks could be abstracted away in the near future. In the end, the application security must be evaluated against the cost to find a good balance.

Chapter 7

Conclusion

Since National Institute of Standards and Technology (NIST)'s endorsement, Advanced Encryption Standard (AES) has become one of the most widely used ciphers. Its simple operations allow for efficient implementation in both software and hardware. Software implementation can take advantage of standard instructions in processors, while hardware accelerators are typically faster but require specialized hardware units. Information leakage through power consumption was discovered in 1999 by *Paul Kocher et al.* [5], which resulted in side-channel attacks that can deduce secret information through statistical analysis. A straightforward implementation of AES is vulnerable to side-channel leakages. The typical countermeasures for AES are threshold implementations and other variants of it.

In this work, a first-order Side-Channel Attack (SCA) secure AES accelerator was developed using the domain-oriented masking scheme for the CV32E40X RISC-V core by the OpenHW group. The Domain-Oriented Masking (DOM) scheme was selected because of its strong foundational basis, the low requirements of randomness, and the generic design that is easy to expand to protect against higher-order SCAs. The accelerator is connected to an Random Number Generator (RNG) that feeds it with the 26 bits of fresh randomness required for each instruction. Eight random bits are used to mask the input and as the second share, and the rest are used to mask the cross-domain products in the Substitution-Box (SBox) multipliers. The simple linear operations of AES are duplicated for each share, while the non-linear SBox must be modified. Some of the subcomponents of the SBox combine data from both shares, such as the multipliers. The result of a function must be non-complete, meaning that it is independent of at least one of the input shares, otherwise leakages could be cause by glitches. Hence, the multipliers must be swapped with the DOM multipliers where cross-domain products are masked by adding fresh-randomness. In the end, the shares are combined to produce the expected result, which means that the inputs and outputs of this accelerator follow the standard defined by the scalar-crypto Instruction Set Extension (ISE).

The accelerator is packed in a wrapper that connects to the CV32E40X core through the extension interface. The wrapper implements logic to accept AES instructions, check if the core has validated the instruction, and write the result back to the core.

The complete accelerator with the wrapper is 2.6 times larger than the unprotected accelerator available in the scalar-crypto ISE repository. Furthermore, the throughput is at most decreased by a factor of 6 for an unoptimized application that cannot take full advantage of the pipeline.

It is crucial that security is considered in every step of the design process. This work focused on passive side-channel attacks, while more invasive attack vectors were not considered. Security measures to mask leakages do not make a difference if an adversary gets access to memory or internal debug features. Furthermore, for security critical components, the chips should be investigated after production to ensure that no malicious modifications have been made. And as reflected in the area and throughput results, adding security is not free. However, simple hardware attacks are becoming ubiquitous, and advances in higher-order attacks result in sound and feasible approaches. It is critical to consider and discuss the level of security required for new technologies because it can be challenging, or even impossible, to protect a device after it has been produced.

7.1 Future Work

The security of the implementation in this work has not been verified due to the limited time aspect. A formal verification method could be performed to test the security of the accelerator by checking if any signals or glitches could leak information. Furthermore, the design could be implemented in an Field-Programmable Gate Array (FPGA) to execute first- and second-order side-channel attacks. The implementation in this work should be secure against first-order and not against second-order attacks. The accelerator could also be extended to be second-order secure to present the ease of extending the design and to discuss the additional cost of securing against higher-orders, since one of the reasons that the DOM scheme was selected in this work was its generic design that is simple to expand.

Additionally, the instruction flow for the AES application could be optimized to improve pipeline utilization.

Appendix A

Composite Field Subcomponents

The irreducible polynomials used in each sub-field were presented in Equation (2.7). The components in composite fields are dependent on the values chosen for the traces and norms in the irreducible polynomials. For a more optimal hardware implementation, the traces are usually set to 1 [30]. Since the irreducible polynomials can not be the same for any subfields, the Norms cannot be set to 1 for

A.1 Operations in GF(2^8)/GF(2^4)

In normal basis, an element in GF(2^8) can be represented in GF($(2^4)^2$) using (Y^{16}, Y) as $A = a_h Y^{16} + a_l Y$, with the irreducible polynomial as shown in Equation (A.1).

$$p(Y) = y^2 + \tau y + v = (y + Y)(y + Y^{16}) = y^2 + (Y + Y^{16})y + YY^{16} \tag{A.1}$$

This gives that the norm $v = (Y)(Y^{16})$ and trace $\tau = y + y^{16} \rightarrow 1 = \tau^{-1}(Y + Y^{16})$. Other useful equations are given in Equation (A.2).

$$\begin{aligned} Y^2 &= \tau Y + v = (Y + Y^{16})Y + YY^{16} = Y^2 \\ Y^{32} &= \tau Y^{16} + v = (Y + Y^{16})Y^{16} + YY^{16} = Y^{32} \end{aligned} \tag{A.2}$$

A.1.1 Inversion

The derivation of $GF(2^8)/GF(2^4)$ inversion is shown in Equation A.3. The inversion property given in Section 2.2.3 has been used as a starting point.

$$
\begin{aligned}
p &= (p_hY^{16} + p_lY) \\
q &= p^{-1} = (q_hY^{16} + q_lY) \\
k = pq &= 1 = (p_hY^{16} + p_lY)(q_hY^{16} + q_lY) \\
&= p_hq_hY^{32} + p_hq_lY^{16}Y + p_lq_hY^{16}Y + p_lq_lY^2 \qquad \mid Y^2 = \tau Y + v \text{ and } Y^{32} = \tau Y^{16} + v \\
&= p_hq_h(\tau Y^{16} + v) + (p_hq_lY + p_lq_h)v + p_lq_l(\tau Y + v) \\
&= p_hq_h\tau Y^{16} + (p_hq_h + p_hq_l + p_lq_h + p_lq_l)v + p_lq_l\tau Y \\
&= p_hq_h\tau Y^{16} + p_lq_l\tau Y + (p_h + p_l)(q_h + q_l)v\tau^{-1}(Y^{16} + Y) \qquad \mid 1 = \tau^{-1}(Y + Y^{16}) \\
&= (p_hq_h\tau + (p_h + p_l)(q_h + q_l)v\tau^{-1})Y^{16} + (p_lq_l\tau(p_h + p_l)(q_h + q_l)v\tau^{-1})Y
\end{aligned}
$$

$$
\begin{aligned}
1 &= \tau^{-1}(Y^{16} + Y) \\
&\Downarrow \\
(1)\text{: } \tau^{-1} &= (p_hq_h\tau + (p_h + p_l)(q_h + q_l)v\tau^{-1}) \\
(2)\text{: } \tau^{-1} &= (p_lq_l\tau + (p_h + p_l)(q_h + q_l)v\tau^{-1}) \\
(1) - (2) &\rightarrow p_hq_h = p_lq_l \\
&\downarrow \\
(p_h + p_l)(q_h + q_l)v\tau^{-1} &= (p_hq_l + p_lq_h)v\tau^{-1}
\end{aligned}
$$

$$
\begin{aligned}
(1) : 1 &= p_hq_h\tau^2 + (p_lq_h + p_hq_l)v \qquad \mid \times p_l \\
p_l &= p_hp_lq_h\tau^2 + (p_l^2q_h + p_hp_lq_l)v \qquad \mid p_lq_l = p_hq_h \\
p_l &= p_hp_lq_h\tau^2 + (p_l^2q_h + p_h^2q_h)v \\
p_l &= q_h(p_hp_l\tau^2 + (p_l^2 + p_h^2)v)
\end{aligned}
$$

$$
\begin{aligned}
q_h &= p_l[p_hp_l\tau^2 + (p_l + p_h)^2v]^{-1} \\
q_l &= p_h[p_hp_l\tau^2 + (p_l + p_h)^2v]^{-1}
\end{aligned}
$$

Equation A.3: Derivation of inversion in composite field $GF(2^8)/GF(2^4)$ for a normal basis

In block three of the equation, the identity, characterized by the Trace, is used to show how the coefficient in front of the normal basis terms (Y^{16}, Y) should be equal to the inverse of the Trace constant.

The resulting two equations consist of five operations in GF(2^4) each, where four out of the five operations can be shared since only the last multiplication differs. There are two multiplications, two additions, and a square-and-scale. Where the last squares the sum and scales the result by the constant Norm v.

Figure 2.5 visualizes the components and connections for the inversion described by Equation A.3. Setting the Trace to 1 is more efficient for hardware implementations; thus, the Trace variable is removed in the figure.

A.2 Operations in GF(2^4)/GF(2^2)

The finite-field $GF(2^8)$ could also be transformed into the composite field $GF((((2)^2)^2)^2)$. An element B from $GF(2^4)$ can be represented in $GF(2^2)$ using (Z^4, Z) as $B = b_hZ^4 + b_lZ$, while the irreducible polynomial as shown in Equation (A.4)

$$s(Z) = z^2 + Tz + N = (z + Z)(z + Z^4) \tag{A.4}$$

$$
\begin{aligned}
Z^2 &= TZ^2 + N = (Z + Z^4)Z + ZZ^4 = Z^2 \\
Z^8 &= TZ^4 + N = (Z + Z^4)Z^4 + ZZ^4 = Z^8
\end{aligned}
\tag{A.5}
$$

$$
\begin{aligned}
k &= k_h Z^4 + k_l Z \\
k &= pq = (p_h Z^4 + p_l Z)(q_h Z^4 + q_l Z) \\
&= p_h q_h Z^8 + (p_h q_l + p_l q_h) Z^5 + p_l q_l Z^2 \qquad | \; Z^8 = TZ^4 + N \text{ and } Z^2 = TZ^2 + N \\
&= p_h q_h (TZ^4 + N) + (p_h q_l + p_l q_h) N + p_l q_l (TZ + N) \\
&= p_h q_h TZ^4 + p_h q_h N + (p_h q_l + p_l q_h) N + p_l q_l TZ + p_l q_l N \\
&= p_h q_h TZ^4 + p_l q_l TZ + (p_h q_h N + p_h q_l + p_l q_h + p_l q_l) N \\
&= p_h q_h TZ^4 + p_l q_l TZ + (p_h + p_l)(q_h + q_l) N (Z^4 + Z) \\
&= (p_h q_h T + (p_h + p_l)(q_h + q_l) N) Z^4 + (p_l q_l T + (p_h + p_l)(q_h + q_l) N) Z \\
&\Downarrow \\
k_h &= p_h q_h T + (p_h + p_l)(q_h + q_l) N \\
k_l &= p_l q_l T + (p_h + p_l)(q_h + q_l) N
\end{aligned}
$$

Equation A.6: Derivation of multiplication in composite field $GF(2^4)/GF(2^2)$ for a normal basis

This gives that the norm $N = (Z)(Z^4)$ and trace $T = Z + Z^4 \rightarrow 1 = T^{-1}(Z + Z^4)$. Other useful equations are given in Equation (A.5).

A.2.1 Multiplier

In a multiplication two terms p and q are multiplied to create the product k. To do this, each of the input terms are split into the subfield $GF(2^2)$. Equation A.6 shows the calculation for this operation.

Figure A.1 shows the multiplicator assuming that Trace T is 1. This multiplication can be implemented using eight subcomponents in $GF(2^2)$: three multiplications, four additions, and one scale by constant N.

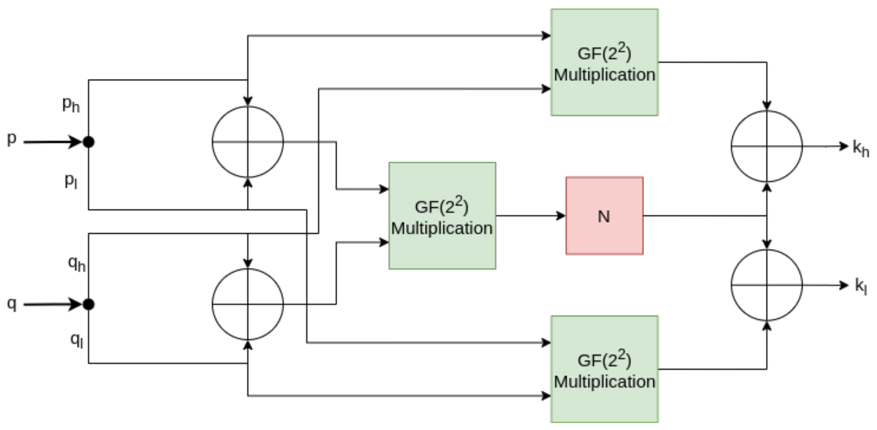

Figure A.1: Multiplicator in composite field $GF(2^4)/GF(2^2)$.

A.2.2 Inversion

The derivation of inversion in $GF(2^4)/GF(2^2)$ is identical to inversion in $GF(2^8)/GF(2^4)$ as shown in Section A.1.1. The only difference is the use of trace T and norm N. Figure A.2 visualizes the inversion in $GF(2^4)/GF(2^2)$. One can quickly see that it has the same structure and components as inversion in $GF(2^8)/GF(2^4)$ just with components of a different subfield.

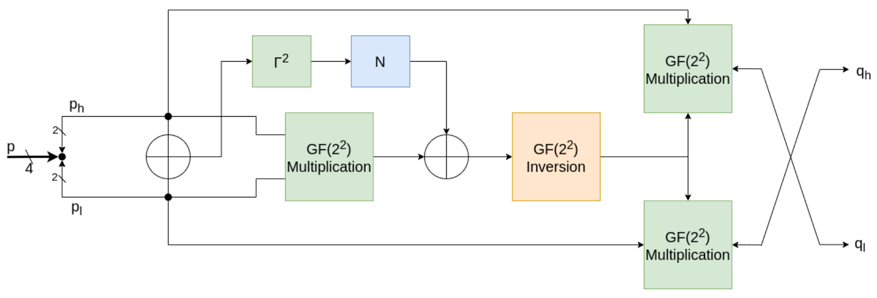

Figure A.2: Composite field inversion for $GF(2^4)$

A.2.3 Square and Scale

The Square-and-Scale is a set of two sequential operations: square and multiply, or scale, by a constant term. In this composite field the squared result is scaled by the constant Norm v. Equation A.8 shows the computation of these two operations.

$$
\begin{aligned}
k &= k_h Z^4 + k_l Z \\
k = q^2 v &= (q_h Z^4 + q_l Z)^2 (v_h Z^4 + v_l Z) \\
&= (q_h^2 Z^8 + q_l^2 Z^2) v \qquad | \; Z^8 = TZ^4 + N \text{ and } Z^2 = TZ + N \\
&= (q_h^2 (TZ^4 + N) + q_l^2 (TZ + N)) v \\
&= (q_h^2 TZ^4 + q_l^2 TZ + (q_h + q_l)^2 N (Z^4 + Z)) v \\
&= (q_h^2 T + (q_h + q_l)^2 N) Z^4 + (q_l^2 T + (q_h + q_l)^2 N) Z) v \\
\hline
A &= q_h^2 T + (q_h + q_l)^2 N \\
B &= q_l^2 T + (q_h + q_l)^2 N \\
\hline
&= (AZ^4 + BZ)(v_h Z^4 + v_l Z) \\
&= A v_h Z^8 + A v_l Z^5 + B v_h Z^5 + b v_l Z^2 \\
&= A v_h (TZ^4 + N) + (A v_l + B v_h) N + b v_l (TZ + N) \\
&= A v_h TZ^4 + (A v_h + A v_l + B v_h + B v_l) N + b v_l TZ \\
&= A v_h TZ^4 + (A + B)(v_h + v_l) N (Z^4 + Z) + b v_l TZ \\
&= (A v_h T + (A + B)(v_h + v_l) N) Z^4 + (b v_l T + (A + B)(v_h + v_l) N) Z \\
\hline
Z^4 \Rightarrow & \\
k_h &= (q_h^2 T + (q_h + q_l)^2 N) v_h T + (q_h^2 + q_l^2)(v_h + v_l) N \\
T = 1 & \\
&= q_h^2 v_h + q_h^2 v_h N + q_l^2 v_h N + q_h^2 v_h N + q_h^2 v_l N + q_l^2 v_h N + q_l^2 v_l N \\
&= q_h^2 v_h + q_h^2 v_l N + q_l^2 v_l N \\
&= q_h^2 v_h + q_h^2 v_l N + q_l^2 v_l N \\
k_h &= q_h^2 v_h + (q_h + q_l)^2 v_l N \\
\hline
Z \Rightarrow & \\
k_l &= (q_l^2 T + (q_h + q_l)^2 N) v_l T + (q_h^2 + q_l^2)(v_h + v_l) N \\
T = 1 & \\
&= q_l^2 v_l + q_h^2 v_l N + q_l^2 v_l N + q_h^2 v_h N + q_h^2 v_l N + q_l^2 v_h N + q_l^2 v_l N \\
&= q_l^2 v_l + q_h^2 v_l N + q_l^2 v_l N \\
&= q_l^2 v_l + q_h^2 v_l N + q_l^2 v_l N \\
k_l &= q_l^2 v_l + (q_h + q_l)^2 v_h N \\
\hline
k_h &= q_h^2 v_h + (q_h + q_l)^2 v_l N \\
k_l &= q_l^2 v_l + (q_h + q_l)^2 v_h N
\end{aligned} \tag{A.8}
$$

Equation A.8: Derivation of square-and-scale in $GF(2^4)/GF(2^2)$

The design of this square-and-scale operation is heavily dependent on the constant norms v and N, and a generic figure will, therefore, not be shown given.

A.3 Operations in GF(2^2)/GF(2)

The lowest subfield of the composite field is GF(2), where the two-bit input is split into two single bits. An element in $GF(2^2)$ can be represented in $GF(2)$ using (W^2, W) as

$C = c_h W^2 + c_l W$ with the irreducible polynomial as shown in Equation (A.9). The Trace and Norm for the prime field are both 1, which gives the equations shown in Equation (A.10)

$$p(w) = w^2 + w + 1 = (w + W)(w + W^2) = w^2 + (W + W^2)w + WW^2 \qquad \text{(A.9)}$$

$$\begin{aligned} N &= 1 = W + W^2 \\ T &= 1 = W^2 W \\ W^4 &= TW^2 + N = (W + W^2)W^2 + WW^2 = W^4 \qquad \text{(A.10)} \end{aligned}$$

A.3.1 Multiplier

Multiplication follows the same calculation as was shown for the multiplier in $GF(2^4)/GF(2^2)$, but the addition operation in multiplication for a field using the prime two can be executed using the Exclusive-OR (XOR) and AND boolean gates, respectively. Figure A.3 shows how the multiplication is built using boolean gates.

$$\begin{aligned} k &= (k_h W^2 + k_l W) \\ k = pq &= (p_h W^2 + p_l W)(q_h W^2 + q_l W) \\ &= (p_h q_h W^4 + p_h q_l W^3 + p_l q_h W^3 + p_l q_l W^2 \qquad | \; W^4 = W \text{ and } W^3 = (W^2 + W) \\ &= p_h q_h W + p_h q_l (W^2 + W) + p_l q_h (W^2 + W) + p_l q_l W^2 \\ &= (p_l q_l + p_h q_l + p_l q_h) W^2 + (p_h q_h + p_h q_l + p_l q_h) W \\ \hline k_h &= p_l q_l + p_h q_l + p_l q_h \\ k_l &= p_h q_h + p_h q_l + p_l q_h \end{aligned}$$

Equation A.11: Derivation of multiplication in composite field $GF(2^2)/GF(2)$ for a normal basis

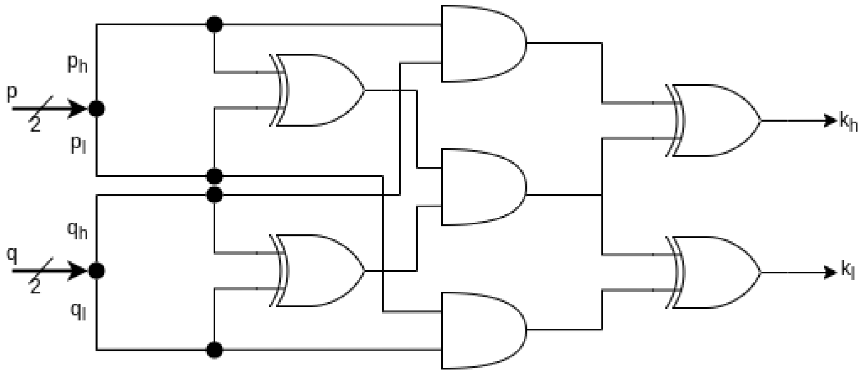

Figure A.3: Multiplication in composite field $GF(2^2)/GF(2)$ using boolean gates

A.3.2 Inversion/Square

There are four elements in $GF(2^2)$: {0, 1, x, x+1}, where there is no multiplicative inverse for the zero element, and the multiplicative inverse for 1 is itself. That means that the last two elements must be each others multiplicative inverse. Thus, inversion and square is the same operation.

$$
\begin{aligned}
k &= k_h W^2 + k_l W \\
k &= q^2 = (q_h W^2 + q_l W)^2 \\
&= q_h^2 W^4 + q_l^2 W^2 && \mid W^4 = W \text{ and } W^2 = W \\
&= q_h^2 W + q_l^2 W^2 && \mid q_h^2 = q_h \text{ and } q_l^2 = q_l \\
&= q_l W^2 + q_h W \\
\hline
k_h &= q_l \\
k_l &= q_h
\end{aligned}
$$

Equation A.12: Derivation of square in $GF(2^2)/GF(2)$

A.3.3 Square and Scale

The square-and-scale operation for $GF(2^2)/GF(2)$ is identical to the one from $GF(2^4)/GF(2^2)$, except for the different constant norm N instead of v. Equation A.14 shows the calculation for this. It is observable that the resulting output vectors are strongly dependent on the constant norm N. Therefore, a figure is not given for the generic implementation.

$$
\begin{aligned}
k &= k_h W^2 + k_l W \\
N &= N_h W^2 + N_l W \\
k = q^2 N &= (q_h W^2 + q_l W)^2 N \\
&= (q_h W^4 + q_l W^2) N \\
&= (q_l W^2 + q_h W)(N_h W^2 + N_l W) \\
&= q_l N_h W^4 + q_l N_l W^3 + q_h N_h W^3 + q_h N_l W^2 && \mid W^4 = W \text{ and } W^3 = W^2 + W \\
&= q_l N_h W + (q_l N_l + q_h N_h)(W^2 + W) + q_h N_l W^2 \\
&= (q_h N_l + q_l N_l + q_h N_h) W^2 + (q_l N_h + q_l N_l + q_h N_h) W && \text{(A.14)} \\
\hline
k_h &= q_h N_l + q_l N_l + q_h N_h \\
k_l &= q_l N_h + q_l N_l + q_h N_h
\end{aligned}
$$

Equation A.14: Derivation of square-and-scale in $GF(2^2)/GF(2)$

Appendix B

Repository Structures

The designs in this work are available in four repositories:

1. Highlevel Model in Python [17]
2. Complete-Accelerator with Wrapper [62]
3. C-Application for AES [65]
4. eXtension Interface Wrapper [73]

B.1 Highlevel Model

The highlevel model [17] was developed in python and includes three implementations:

1 Standard AES

2 Scalar-crypto AES implementation

3 2-share DOM AES implementation

The standard Advanced Encryption Standard (AES) is a straightforward highlevel implementation of AES as it could be implemented in software. The Scalar-crypto AES implementation follows the design described in Section 4.1.1. It includes a composite field Substitution-Box (SBox) as described in Section 2.3.2. The last implementation extends the Scalar-crypto implementation to a two share version, which includes a DOM SBox.

B.2 Complete Accelerator With Wrapper

There are four HDL files in the src folder in this repository [62]

1 AES_protected

2 DOM_sbox

3 XIF_AES_wrapper

4 FIFO

The AES_protected contains the 2-share accelerator logic as shown in Section 5.3. The DOM_SBox implements a pipelined domain oriented masking SBox as was described in Section 5.2. The XIF_AES_wrapper contains the eXtension Interface described in Section 5.4, which includes the FIFO.

B.3 C-Application

There is one AES file in this repository [65] that implements the four main functions described in Section 5.5.

B.4 eXtension Interface

A standalone repository [73] has been made for the eXtension interface wrapper. The generic design could easily be adapted for other accelerators that want to connect to the XIF.

Bibliography

[1] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.

[2] National Institute of Standards and Technology. Advanced encryption standard. *NIST FIPS PUB 197*, 2001. (Accessed on 08/03/2022).

[3] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012.

[4] Joan Daemen and Vincent Rijmen. The wide trail design strategy. In Bahram Honary, editor, *Cryptography and Coding*, pages 222–238, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[5] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[6] ChipWhisperer. Chipwhisperer training. `https://learn.chipwhisperer.io/`. (Accessed on 05/28/2022).

[7] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power analysis attacks of modular exponentiation in smartcards. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems*, pages 144–157, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[8] 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' 99*, pages 398–412, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[9] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, pages 529–545, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[10] Hannes Groß. *Domain-Oriented Masking: Generically Masked Hardware Implementations*. PhD thesis, Graz University of Technology (90000), 2018.

[11] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. Cryptology ePrint Archive, Paper 2015/719, 2015. `https://eprint.iacr.org/2015/719`.

[12] Hannes Gross and Stefan Mangard. A unified masking approach. *Journal of Cryptographic Engineering*, 8, 06 2018.

[13] NIST. Time to standardize threshold schemes for cryptographic primitives | csrc. `https://csrc.nist.gov/publications/detail/itl-bulletin/2019/04/standardize-threshold-schemes-for-cryptographic-primitives/final`, April 2019. (Accessed on 06/07/2022).

[14] OpenHW. About us | openhw group. `https://www.openhwgroup.org/about-us/`. (Accessed on 06/12/2022).

[15] OpenHW. openhwgroup/core-v-cores: Core-v family of risc-v cores. `https://github.com/openhwgroup/core-v-cores`. (Accessed on 03/30/2022).

[16] OpenHW. riscv/riscv-crypto: Risc-v cryptography extensions standardisation work. `https://github.com/riscv/riscv-crypto`. (Accessed on 01/25/2022).

[17] Aleksander Waage. Aes-highlevel-model repository. `https://github.com/Aleksw3/AES-Highlevel-Model`. (Accessed on 06/10/2022).

[18] OpenHW Group. Openhw group core-v verification strategy — core-v verification strategy documentation. `https://core-v-docs-verif-strat.readthedocs.io/en/latest/`. (Accessed on 05/28/2022).

[19] NIST. Roadmap: Criteria for threshold schemes for crypto primitives | csrc. `https://csrc.nist.gov/publications/detail/nistir/8214a/final`, July 2020. (Accessed on 05/27/2022).

[20] Christof Paar and Jan Pelzl. *The Data Encryption Standard (DES) and Alternatives*, pages 55–86. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[21] Mukta Sharma and R.B. Garg. Des: The oldest symmetric block key encryption algorithm. In *2016 International Conference System Modeling Advancement in Research Trends (SMART)*, pages 53–58, 2016.

[22] Joan Daemen and Vincent Rijmen. *The Design of Rijndael.* Springer-Verlag, Berlin, Heidelberg, 2002.

[23] NIST. Announcing request for candidate algorithm nominations for the advanced encryption standard. `https://www.govinfo.gov/content/pkg/FR-1997-09-12/pdf/97-24214.pdf`. (Accessed on 06/13/2022).

[24] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and E Roback. Report on the development of the advanced encryption standard (aes), 2001-06-01 2001.

[25] Claude Shannon. A mathematical theory of cryptography, 1945.

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

[27] Elisabeth Oswald, Joan Daemen, and Vincent Rijmen. Aes-the state of the art of rijndael's security. *Prieiga internetu: http://books. google. lt/books/about/The_Design_of_Rijndael. html*, 2002.

[28] Shuhong Gao. *Normal bases over finite fields.* University of Waterloo Waterloo, Canada, 1993.

[29] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient rijndael encryption implementation with composite field arithmetic. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 171–184, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[30] D. Canright. A very compact s-box for aes. In *Proceedings of the 7th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'05, page 441–455, Berlin, Heidelberg, 2005. Springer-Verlag.

[31] Marcel Berger. *Geometry i.* Springer Science & Business Media, 2009.

[32] Joan Boyar and Rene Peralta. A depth-16 circuit for the aes s-box. Cryptology ePrint Archive, Report 2011/332, 2011. `https://ia.cr/2011/332`.

[33] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[34] Taras Iakymchuk, Maciej Nikodem, and Krzysztof Kępa. Temperature-based covert channel in fpga systems. In *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, 2011.

[35] Yu-ichi Hayashi, Naofumi Homma, Takaaki Mizuki, Haruki Shimada, Takafumi Aoki, Hideaki Sone, Laurent Sauvage, and Jean-Luc Danger. Efficient evaluation of em radiation associated with information leakage from cryptographic devices. *IEEE Transactions on Electromagnetic Compatibility*, 55(3):555–563, 2013.

[36] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective.* Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[37] Shen, Ghosh, Devadas, and Keutzer. On average power dissipation and random pattern testability of cmos combinational logic networks. In *1992 IEEE/ACM International Conference on Computer-Aided Design*, pages 402–407, 1992.

[38] Thomas Korak and Michael Hoefler. On the effects of clock and power supply tampering on two microcontroller platforms. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 8–17, 2014.

[39] Mark Randolph and William Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2), 2020.

[40] Owen Lo, William J. Buchanan, and Douglas Carson. Power analysis attacks on the aes-128 s-box using differential power analysis (dpa) and correlation power analysis (cpa). *Journal of Cyber Security Technology*, 1(2):88–107, 2017.

[41] 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*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[42] 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*, pages 192–207, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[43] Benedikt Gierlichs, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. Revisiting higher-order dpa attacks:. In Josef Pieprzyk, editor, *Topics in Cryptology - CT-RSA 2010*, pages 221–234, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[44] Huanyu Wang and Elena Dubrova. Tandem deep learning side-channel attack against fpga implementation of aes. In *2020 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS)*, pages 147–150, 2020.

[45] Richard Gilmore, Neil Hanley, and Maire O'Neill. Neural network based attack on a masked implementation of aes. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 106–111, 2015.

[46] Suresh Chari, Charanjit Jutla, Josyula R. Rao, and Pankaj Rohatgi. A cautionary note regarding evaluation of aes candidates on smart-cards. In *In Second Advanced Encryption Standard (AES) Candidate Conference*, pages 133–147.

[47] Eng. Mustafa M. Shiple, Prof. Dr. Iman S. Ashour, and Prof. Dr. Abdelhady A. Ammar. Attacking misaligned power tracks using fourth-order cumulant. *International Journal of Advanced Computer Science and Applications*, 4(12), 2013.

[48] Amos Beimel. Secret-sharing schemes: A survey. In Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing, editors, *Coding and Cryptology*, pages 11–46, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[49] 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*, pages 309–318, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[50] 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*, pages 351–365, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[51] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia Tokareva, and Valeriya Idrisova. Threshold implementations of small s-boxes. *Cryptography and Communications*, 7, 03 2015.

[52] RISC-V. The risc-v instruction set manual - volume i: Unprivileged isa. `https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf`, December 2019. (Accessed on 03/29/2022).

[53] Wikipedia. Risc-v - wikipedia. `https://en.wikipedia.org/wiki/RISC-V#Design`. (Accessed on 03/29/2022 and used because Wikipedia had the most up to date ISE collection).

[54] RISC-V. "risc-v cryptography extensions volume i: Scalar & entropy source instructions". `https://github.com/riscv/riscv-crypto/releases/tag/v1.0.0-scalar`. (Accessed on 01/31/2022).

[55] grnewell. Shorter names, and consistency for instructions with immediate operands · issue #46 · riscv/riscv-crypto. `https://github.com/riscv/riscv-crypto/issues/46`, September 2020. (Accessed on 06/08/2022).

[56] Risc-V. "risc-v exchange: Cores & socs - risc-v international". `https://riscv.org/exchanges/cores-socs/`. (Accessed on 03/30/2022).

[57] OpenHW. 4 stage, in-order, secure risc-v core based on the cv32e40p. `https://github.com/openhwgroup/cv32e40s`. (Accessed on 06/13/2022).

[58] OpenHW. Cv32e40x proposal. `https://github.com/openhwgroup/core-v-docs/blob/master/program/Project%20Descriptions%20and%20Plans/CV32E40X/CV32E40X-PPL.md`. (Accessed on 02/21/2022).

[59] OpenHW. openhwgroup/cv32e40x: 4 stage, in-order, compute risc-v core based on the cv32e40p. `https://github.com/openhwgroup/cv32e40x`. (Accessed on 03/30/2022).

[60] OpenHW. Core-v extension interface documentation. `https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/intro.html`. (Accessed on 02/19/2022).

[61] Core-v cv32e40x user manual documentation. `https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/intro.html`. (Accessed on 02/21/2022).

[62] Aleksander Waage. Hdl aes-scalar-crypto repository. `https://github.com/Aleksw3/AES-Scalar-Crypto`. (Accessed on 06/10/2022).

[63] Tool chain downloads – embecosm. `https://www.embecosm.com/resources/tool-chain-downloads/#corev`. (Accessed on 05/11/2022).

[64] Lee More and Duncan Graham. A guide to accelerating applications with just-right risc-v custom instructions. `https://www.embedded.com/a-guide-to-accelerating-applications-with-just-right-risc-v-custom-instructions/`, June 2020. (Accessed on 06/08/2022).

[65] Aleksander Waage. Aes-c-application repository. `https://github.com/Aleksw3/AES-C-application`. (Accessed on 06/10/2022).

[66] Nils Wiersma and Ramiro Pareja. Safety != security: On the resilience of asil-d certified microcontrollers against fault injection attacks. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 9–16, 2017.

[67] Disruptive Technologies. Security and privacy in disruptive technologies iot sensing solution. `https://www.disruptive-technologies.com/hubfs/Help-Center/DT-White-Paper-Security.pdf`. (Accessed on 05/28/2022).

[68] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of aes. In *Advances in Cryptology – EUROCRYPT 2011*, pages 69–88. Springer Berlin Heidelberg, 2011.

[69] Begül Bilgin. *Threshold implementations: as countermeasure against higher-order differential power analysis.* PhD thesis, University of Twente, Netherlands, May 2015. Cum laude.

[70] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[71] Lang Lin, Markus Kasper, Tim Güneysu, Christof Paar, and Wayne Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 382–395, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[72] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor. Hardware trojans: Lessons learned after one decade of research. *ACM Trans. Des. Autom. Electron. Syst.*, 22(1), may 2016.

[73] Aleksander Waage. Aleksw3/extension-inteface-wrapper. `https://github.com/Aleksw3/eXtension-Inteface-Wrapper`, June 2022. (Accessed on 06/12/2022).