KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

A White-Box Speck Implementation using Self-Equivalence Encodings

Joachim Vandersmissen

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdoptie Veilige software

Promotoren:
Prof. dr. ir. F. Piessens
Prof. dr. ir. B. Preneel

Assessor:
Dr. A. Rafique

Begeleider:
A. Ranea

Academiejaar 2020 – 2021

Preface

In the first place, I would like to thank my mentor, Adrián Ranea, for patiently explaining all new concepts and answering my many questions. Even though the circumstances were not ideal this year, I very much enjoyed discussing my work during our video calls. On top of this, Adrián provided very valuable feedback on this text. It would not have been possible to finish this thesis without his help.

I would also like to express my gratitude to professor Preneel, for giving me the opportunity to discover this interesting topic, and to professor Piessens, for allowing me to start this thesis under his supervision and helping me with practical difficulties.

Lastly, I would like thank to my parents for supporting me. Throughout the last five years, they have always had my back.

Joachim Vandersmissen

Contents

Abstract

White-box cryptography is used to protect cryptographic keys in implementations against adversaries with full control over the environment in which they are executed. In 2002, Chow et al. initiated the formal study of white-box cryptography and introduced the CEJO framework. Since then, various other white-box designs have been proposed. However, at the time of writing, there has been no public academic research into white-box cryptography for add-rotate-xor (ARX) ciphers.

In this thesis, we introduce the first academic method to generate white-box Speck implementations. Unlike the CEJO framework, which is based on small random permutations, we use self-equivalences of the Speck substitution layer to encode the round keys. If these self-equivalences are chosen at random, it should be impossible to recover the keys without knowing the original self-equivalences. While we focus on Speck in this text, our method could easily be adapted to protect other ARX ciphers.

Then, we analyze the security of our method against key-recovery attacks. We discover that, when only linear self-equivalences are used, only one round key bit is hidden by the self-equivalence encoding. Furthermore, we propose an algebraic attack to fully recover the linear self-equivalence encodings and external encodings from a white-box Speck implementation. When affine self-equivalences are used, we show that candidate round keys could be computed by guessing the values of two key bits. However, we are not able to recover the external encodings when affine self-equivalences are used. Still, the security of our method remains uncertain. We hope that this work will spur additional research in self-equivalence encodings for white-box cryptography.

Finally, we describe the architecture of a Python project implementing our method. This implementation is publicly available at https://github.com/jvdsn/white-box-speck. We explain various design decisions, and use this project to compute the performance impact of our method. Moreover, we give an overview of five additional strategies to generate output code. These strategies can be used to improve the performance of the generated code, both in terms of disk space usage and execution time. We compare these strategies and determine how to generate the most performant white-box Speck code. Lastly, this project could be employed to test and compare the efficiency of attacks on white-box implementations using self-equivalence encodings. For example, we use this project to demonstrate our attack on white-box Speck using only linear self-equivalences.

Samenvatting

White-box cryptography wordt gebruikt om cryptografische sleutels in implementaties te beschermen tegen aanvallers die complete controle hebben over de uitvoeringsomgeving. In 2002 begonnen Chow et al. de formele studie van white-box cryptography en introduceerden ze het CEJO raamwerk. Sindsdien zijn er ook verschillende andere white-box ontwerpen voorgesteld. Echter is er, tot op heden, nog geen academisch onderzoek naar white-box cryptografie voor *add-rotate-xor* (ARX) cijfers gevoerd.

In deze thesis introduceren we de eerste academische methode om white-box Speck implementaties te genereren. In tegenstelling tot het CEJO raamwerk, dat gebaseerd is op kleine willekeurige permutaties, gebruiken we *self-equivalences* van de Speck substitutielaag om de ronde sleutels te coderen. Als deze self-equivalences willekeurig gekozen worden, zou het onmogelijk moeten zijn om de sleutels zonder voorkennis te achterhalen. Alhoewel we in deze tekst focussen op Speck, kan onze methode makkelijk aangepast worden om ook andere ARX cijfers te beschermen.

Daarna analyseren we de veiligheid van onze methode tegen *key-recovery* aanvallen. We ontdekken dat, wanneer enkel lineaire self-equivalences gebruikt worden, slechts één bit van de ronde sleutel verborgen wordt door de self-equivalence codering. Daarbovenop stellen we een algebraïsche aanval voor om de lineaire self-equivalence coderingen én externe coderingen te achterhalen uit een white-box Speck implementatie. Wanneer affiene self-equivalences gebruikt worden, tonen we dat kandidaat ronde sleutels berekend kunnen worden door de waarde van twee bits te raden. We waren echter niet in staat om in dit geval de externe coderingen achterhalen. Toch blijft de veiligheid van onze methode onzeker. We hopen dat dit werk een aanzet is naar meer onderzoek in self-equivalence coderingen voor white-box cryptografie.

Ten slotte beschrijven we de architectuur van een Python project dat onze methode implementeert. Deze implementatie is publiek beschikbaar op `https://github.com/jvdsn/white-box-speck`. We verklaren verschillende ontwerpkeuzes, en gebruiken dit project om de prestatie-impact van onze methode te berekenen. Daarnaast geven we een overzicht van vijf bijkomende strategieën om uitvoercode te genereren. Deze strategieën kunnen gebruikt worden om de prestaties van de gegenereerde code, zowel op vlak van schijfgebruik als uitvoeringstijd, te verbeteren. We vergelijken deze strategieën en bepalen hoe de meest performante white-box Speck code gegenereerd kan worden. Tot slot kan dit project aangewend worden om aanvallen op white-box implementaties met self-equivalence coderingen te testen. We gebruiken dit project bijvoorbeeld om onze aanval op white-box Speck implementaties met enkel lineaire self-equivalence coderingen te demonstreren.

List of Figures and Tables

List of Figures

List of Tables

Chapter 1

Introduction

Traditionally, honest parties use cryptographic algorithms in combination with cryptographic keys to encrypt or decrypt messages. However, there are situations in which these keys must remain hidden, even from the party performing the encryption or decryption. In this case, the adversary has full control over the execution environment. As such, the implementation is a "white box" to the adversary. White-box cryptography is used to protect these implementations against key recovery attacks. From an attacker's perspective, reverse engineering and extracting a protected implementation of a cipher is less convenient compared to simply redistributing the keys. This implementation might also be restricted to a specific computing platform. Consequently, white-box cryptography is a popular method to protect private keys in the mobile banking industry and for digital rights management (DRM).

In 2002, Chow et al. initiated the formal study of white-box cryptography in their seminal work [1]. They provided a formal definition of the white-box model, which specifies the capabilities of an adversary with complete control of a cryptographic implementation. On top of this, they introduced the first academic framework to generate protected implementations in the white-box model, based on the AES block cipher. Since then, many different constructions have been developed over the years. Nevertheless, all of the block cipher implementations using the approach of Chow et al. have been broken.

White-box cryptography is closely related to the fields of program obfuscation in software development and cryptographic obfuscation. The goal of obfuscation is to hide some information about the description of a program, without changing the behavior of said program. In software development, this could be achieved by changing variable names or introducing complicated control flow, whereas cryptographic obfuscation is based on mathematical problems. In 2001, Barak et al. proved that it is impossible to perfectly obfuscate all programs (black-box obfuscation) [2]. Moreover, applying techniques from cryptographic obfuscation to complex functions, such as block ciphers, is currently not practical. Still, research in black-box obfuscation and other models, such as indistinguishability obfuscation, continues.

In this thesis, we will try to apply white-box protection to the Speck family of block ciphers, based on add-rotate-xor (ARX) ciphers. At the time of writing this

text, there is no public academic research on ARX ciphers in the context of white-box cryptography. However, some commercial products, such as whiteCryption's *Secure Key Box* [3], do offer this possibility. Unfortunately, companies rarely publish their methods for white-box cryptography, and the security of these implementations depends on the secrecy of their design.

The goal of this thesis is threefold. Firstly, we want to introduce a method to generate secure white-box implementations for Speck encryption. This method will be based on self-equivalence encodings, introduced by McMillion et al. [4] and recently picked up by Ranea et al. [5]. A self-equivalence of a function is a pair of permutations which can be applied to the start and end of the function without changing the original behavior. Random self-equivalences of the Speck substitution layer could be applied to the start and end of each affine layer without changing the Speck behavior. It should then be infeasible to recover the round keys embedded in these encoded layers, without knowing the original self-equivalence encodings.

Secondly, we want to conduct a security analysis of our method in the white-box model. One important question is whether the round keys are adequately masked by the self-equivalence encodings. Additionally, we want to investigate whether it could be possible to recover the original self-equivalence encodings from the encoded affine layers. Recovering these encodings would allow an attacker to reverse the protection and recover the embedded keys.

Finally, we want to create a functional implementation of this method, capable of generating correct white-box Speck code. Most importantly, this would allow us to compare the performance impact of our method to an unprotected Speck implementation. As we expect this impact to be significant, we could extend the program with strategies to generate more performant protected code. Moreover, a working implementation would allow us to easily test any key recovery attacks found during our security analysis of our method. Lastly, by publishing this project, we encourage additional cryptanalysis of our method using techniques that might not be covered in this thesis.

The rest of this thesis is structured as follows. In Chapter 2, we define some preliminary notation and concepts that will be reused throughout this text. Furthermore, we give an overview of the Speck block cipher family and its parameter sets. In Chapter 3, we take a closer look at the origins of white-box cryptography, the white-box model, and the CEJO framework. We also describe the state of the art in white-box attacks and countermeasures. Then, in Chapter 4, we introduce the mathematical concept of self-equivalences, our approach to apply self-equivalence encodings to Speck, and a method to efficiently generate these self-equivalences. In Chapter 5, we will analyze the security of our white-box Speck implementation using self-equivalence encodings. Finally, in Chapter 6, we give an overview of our Python project to generate white-box Speck implementations using self-equivalence encodings. This chapter also contains a comparison of five additional strategies to improve the performance of the generated programs, both in terms of disk space usage and execution time.

Chapter 2

Preliminaries

In this chapter, we introduce the most important notation and concepts used in this thesis. We start with an overview of the notation and terminology in Section 2.1. Afterwards, we introduce the SPECK block cipher family in Section 2.2.

2.1 Notation

In general, lowercase symbols in this thesis refer to numbers and vectors, while uppercase symbols are used to denote functions and matrices. In particular, E and D will be used to denote encryption and decryption functions, respectively. On top of this, we use E_k and D_k to refer to encryption and decryption functions with a hard coded key, k.

Finite fields with q elements are written as $\mathbb{F}_q$. We will only work with the finite field over two elements, $\mathbb{F}_2$. Vectors over this field are called binary vectors, while matrices over $\mathbb{F}_2$ are called binary matrices. More specifically, binary vectors in the vector space $\mathbb{F}_2^n$ are called n-bit vectors. The addition in $\mathbb{F}_2$ is denoted using $\oplus$, and we extend this to the addition of n-bit vectors by pairwise addition of each element. Finally, as a shorthand, we will sometimes replace $\oplus\, c$ by $\oplus_c$ if c is a constant.

A function $A : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ is called an (n, m)-bit function. If $n = m$, then we simply call these functions n-bit functions. We use $\circ$ to refer to the composition of functions and $||$ to refer to the concatenation of functions. For example, $(A||B)(x, y) = (A(x), B(y))$.

We generally use calligraphic font to indicate sets, for example $\mathcal{A}$. The cardinality of $\mathcal{A}$ is denoted using $|\mathcal{A}|$. We can also extend the function composition and function concatenation operations to sets of functions as follows:

$$\mathcal{A} \circ \mathcal{B} = \{A \circ B : A \in \mathcal{A}, B \in \mathcal{B}\}$$
$$\mathcal{A}||\mathcal{B} = \{A||B : A \in \mathcal{A}, B \in \mathcal{B}\}$$

Moreover, if $\mathcal{A}$ contains only a single element A, we simply write $A \circ \mathcal{B}$ or $A||\mathcal{B}$.

An important operation in this thesis is the *modular addition*, defined as the addition of two numbers x and y, modulo some power of two. We use $\boxplus$ to refer to the modular addition, and $\boxminus$ to refer to its inverse, the *modular subtraction*.

Furthermore, $x \ggg \alpha$ denotes a right bitwise circular shift of x by α positions and $x \lll \beta$ denotes a left bitwise circular shift of x by β positions. Finally, in some places we borrow notation from the C programming language: $x\&y$ refers to the bitwise AND operation of the numbers x and y.

2.2 Speck

Speck is a family of lightweight block ciphers proposed by the National Security Agency in 2013 [6]. In particular, Speck was designed with a focus on performance in software. In this thesis, we also use "the Speck (block) cipher" to refer to the general design of the Speck family. As with many block ciphers, the Speck design is based on iterated application of so-called *round functions* on some internal state.

The Speck family consists of ten different instances, depending on the *block size* and *key size* parameters (Table 2.1). The block size refers to the size in bits of the input, internal state, and output. These values always consist of two words, x and y, with bit size n. The key size refers to the size in bits of the master key k, which consists of m key words, with bit size n. We use the block size and key size in a shorthand notation to refer to specific Speck instances. For example, Speck128/256 refers to the Speck instance with block size 128 and key size 256.

Block size	**Key size**	**Word size** n	**Key words** m	**Rounds** n_r	α	β
32	64	16	4	22	7	2
48	72	24	3	22	8	3
48	96	24	4	23	8	3
64	96	32	3	26	8	3
64	128	32	4	27	8	3
96	96	48	2	28	8	3
96	144	48	3	29	8	3
128	128	64	2	32	8	3
128	192	64	3	33	8	3
128	256	64	4	34	8	3

Table 2.1: Speck parameter sets

Algorithm 1 Speck encryption round function

$x, y \leftarrow$ encryption round input
$x \leftarrow x \ggg \alpha$
$x \leftarrow x \boxplus y$
$x \leftarrow x \oplus k^{(r)}$
$y \leftarrow y \lll \beta$
$y \leftarrow x \oplus y$
return x, y

Let $k^{(r)}$ be the round key for round r, and α, β the parameters as defined in Table 2.1. Then the algorithm to compute the round function $E^{(r)}$ for round r is described in Algorithm 1.

Figure 2.1 shows a schematic representation of the SPECK round function.

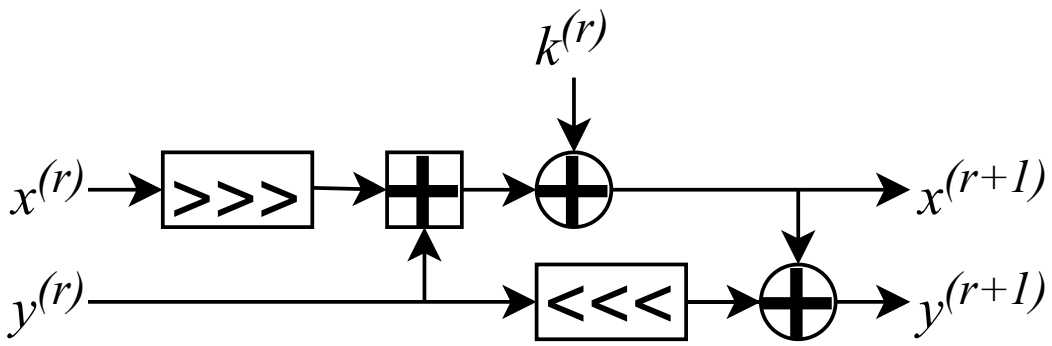

FIGURE 2.1: Diagram of a SPECK encryption round

The decryption round function, $D^{(r)}$, can be computed by simply inverting the steps in Algorithm 1. In particular, the modular addition $\boxplus$ must be replaced by the modular subtraction $\boxminus$.

The round keys, $k^{(r)}$ for $1 \leq r \leq n_r$, are computed using the SPECK key schedule. The key schedule transforms the master key k into n_r round keys using the encryption round function. This allows SPECK implementations to reuse this subroutine, thus reducing code size. We start by writing k as a sequence of m key words ($2 \leq m \leq 4$):

$$k = \left[\begin{array}{cccc} l^{(m-1)} & \dots & l^{(1)} & k^{(1)} \end{array} \right]$$

Then, $l^{(r+m-1)}$ and $k^{(r+1)}$ are computed using the SPECK round function, but with r as "round key":

$$\begin{aligned} l^{(r+m-1)} &= (k^{(r)} \boxplus (l^{(r)} \ggg \alpha)) \oplus r \\ k^{(r+1)} &= (k^{(r)} \lll \beta) \oplus l^{(r+m-1)} \end{aligned}$$

Chapter 3

White-box cryptography

In this chapter, we give a brief overview of the field of white-box cryptography. We start by introducing the white-box model and its main properties in Section 3.1. In Section 3.2, we look at the CEJO framework, one of the most popular methods to generate white-box implementations. Finally, we describe the state of the art in white-box attacks and countermeasures in Section 3.3.

3.1 The White-Box Attack Context

Traditionally, the study of cryptography was mostly constrained to the protection of communications between two honest parties. We call these parties *Alice* and *Bob*. Alice and Bob can use cryptography to securely exchange information without allowing an adversary to read or modify the message while it is transmitted over an insecure channel. To accomplish this, they decide on a cryptographic scheme and the cryptographic keys to use in advance. They can then use the scheme and the keys to encrypt or decrypt some input messages to be transmitted over the insecure channel. In this thesis we will only consider symmetric cryptography, which implies the same key can be used to both encrypt and decrypt data. By Kerckhoffs's principle, the cryptographic scheme is publicly known. As a result, the problem of securing communications is shifted to the exchange of the cryptographic keys.

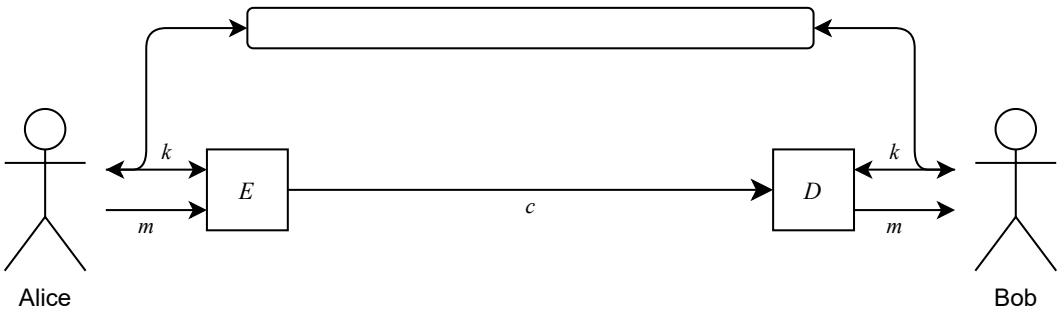

FIGURE 3.1: Securing information using traditional symmetric cryptography

Figure 3.1 shows an example of this scenario. Firstly, Alice and Bob exchange the key k using a secure channel. Then, Alice uses k and the encryption function E to encrypt the message m. E takes a key k and m and produces a ciphertext c, such that it is infeasible to recover m from c without k. The resulting ciphertext can then be sent to Bob, who uses k and the decryption function D to recover the original message m.

In the previous scenario, Alice and Bob are trusted parties to the cryptographic scheme. There is no incentive to cheat this scheme, as their end goal is to protect their private communications against external adversaries. However, this is not always the case in real-world situations. For example, a credit card uses cryptography to enable secure purchases, but also restricts the usage of the card by the customer. Similarly, a video streaming platform might want to send some encrypted videos to a paying customer, but simultaneously prevent this customer from decrypting other videos without permission. The challenge here is to design a cryptographic scheme which is secure against adversarial parties with complete control of the implementation of the scheme.

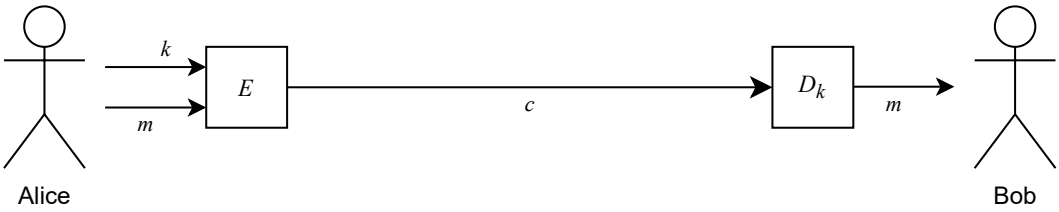

Figure 3.2: Securing information using symmetric cryptography in the white-box model

An example of this scenario is illustrated in Figure 3.2. Alice has access to a cryptographic key k and an encryption function E, and wants to securely send a message m to Bob. However, she does not want Bob to recover k (for example, Alice wants to prevent Bob from using k in a different decryption implementation). Alice can send a white-box protected version of the decryption function called D_k, which can decrypt ciphertexts without revealing k. Then, Bob simply uses D_k to decrypt c from Alice and recover m. Crucially, Bob has full control over the environment in which D_k is executed.

This scenario was formalized in 2002 by Chow et al. in their seminal work [1]. They introduced the *White-Box Attack Context*, also called the *white-box model*. The white-box model has three main properties:

- The attacker is a privileged user on the same host as the cryptographic algorithm, with complete access to the implementation.
- The attacker can dynamically execute the cryptographic algorithm.
- At any point before, during, or after the execution, the attacker is able to view and modify the internal details of the implementation.

In this model, the goal of the attacker is generally to extract the cryptographic keys from the algorithm implementation. This would allow the attacker to execute the encryption or decryption algorithm independently, without relying on the provided implementation. Chow et al. described two approaches to protect cryptographic implementations.

The first approach is called the *dynamic-key approach*. The cryptographic keys are encrypted or encoded in some way to prevent the attacker from recovering the original values. Both the encoded keys and the input data are provided to the protected algorithm. In other words, E accepts an encoded key $\overline{k}$ and m, and produces the ciphertext c. Similarly, D takes the encoded key $\overline{k}$ and c to recover m. This approach has seen less adoption, as it appears to be more difficult than the second approach.

Alternatively, the cryptographic keys can be embedded in the implementation. This is called the *fixed-key approach*. In this case, the keys are part of the algorithm, and only one input is provided to the algorithm: the data to be encrypted or decrypted. For example, E_k encrypts a message using the embedded key k, and D_k decrypts a ciphertext using k. Of course, this reduces the flexibility of the implementation, as the cryptographic key cannot be changed easily. After all, key rotation is an important technique to maintain security in applied cryptography. Nonetheless, Chow et al. argue that, in software, keys can still be reasonably rotated by simply changing the entire implementation.

The fixed-key approach is a very popular way to implement white-box cryptography in practice. Especially in industries such as mobile banking or digital rights management (DRM), commercial applications often employ white-box cryptography to protect cryptographic keys. Thales' *Sentinel* [7], Irdeto's *Cloakware* [8], and whiteCryption's *Secure Key Box* [3] are among the most prolific products offering white-box protection to customers. However, the security of these implementations mainly relies on the secrecy of their design. Consequently, the initiative to develop secure and open methods for white-box cryptography must come from the (academic) research community. In this thesis, we will propose a method using the fixed-key approach.

3.2 The CEJO framework

In their work, Chow et al. also proposed the first academic white-box cryptographic scheme, based on the AES block cipher [9]. Their scheme, also called the CEJO framework, composes the round function with randomized input and output encodings. To showcase the similarities between the CEJO framework and the self-equivalence encodings used in this thesis, introduced in the next chapter, we briefly explain the CEJO framework in this section. For more information, we refer to the original work [1] and a simplified tutorial [10].

3.2.1 Lookup tables

The first step in the CEJO framework to protect an AES encryption function with n_r rounds, $E_k = E^{(n_r)} \circ \cdots \circ E^{(2)} \circ E^{(1)}$, is implementing each encryption round $E^{(r)}$ using only lookup tables. We start by writing $E^{(r)}$ as follows:

$$E^{(r)} = \texttt{MixColumns} \circ \texttt{SubBytes} \circ \texttt{AddRoundKey}_{\hat{k}^{(r-1)}} \circ \texttt{ShiftRows}$$
$$E^{(n_r)} = \texttt{AddRoundKey}_{k^{(n_r)}} \circ \texttt{SubBytes} \circ \texttt{AddRoundKey}_{\hat{k}^{(n_r-1)}} \circ \texttt{ShiftRows}$$

Here, `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey` are the well-known AES operations. Although this description is equivalent to the conventional description of the AES rounds, some notational changes have been made to easily construct a white-box version of the cipher. Firstly, the order of the operations is changed, with `SubBytes` and `ShiftRows` swapped. On top of this, `AddRoundKey` is placed after the `ShiftRows` operation, by replacing the round key $k^{(r)}$ with $\hat{k}^{(r)}$, which represents the result of `ShiftRows` applied to the original round key. These changes allow us to combine the `AddRoundKey` and `SubBytes` operations in new lookup tables, called *T-boxes*.

Definition 1 (T-box)**.** *Let S be the AES S-box, and $\hat{k}_{i,j}^{(r-1)}$ the byte of the round key $\hat{k}^{(r-1)}$ at index (i,j). Then $T_{i,j}^{(r)}$ is defined as follows:*

$$T_{i,j}^{(r)}(x) = S(x \oplus \hat{k}_{i,j}^{(r-1)})$$
$$T_{i,j}^{(n_r)}(x) = S(x \oplus \hat{k}_{i,j}^{(n_r-1)}) \oplus \hat{k}_{i,j}^{(n_r)}$$

This results in $16n_r$ T-boxes, which replace the `AddRoundKey` and `SubBytes` steps in $E^{(r)}$.

The next step is to provide lookup tables for the `MixColumns` operation. Recall that `MixColumns` can be implemented by multiplying a 4×4 matrix and a 4×1 vector for each column in the AES state matrix. This multiplication can be decomposed into four scalar products joined by three XOR operations:

$$y = \begin{bmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = x_1 \begin{bmatrix} \texttt{02} \\ \texttt{01} \\ \texttt{01} \\ \texttt{03} \end{bmatrix} \oplus x_2 \begin{bmatrix} \texttt{03} \\ \texttt{02} \\ \texttt{01} \\ \texttt{01} \end{bmatrix} \oplus x_3 \begin{bmatrix} \texttt{01} \\ \texttt{03} \\ \texttt{02} \\ \texttt{01} \end{bmatrix} \oplus x_4 \begin{bmatrix} \texttt{01} \\ \texttt{01} \\ \texttt{03} \\ \texttt{02} \end{bmatrix}$$

These scalar products are then replaced by lookup tables, the so-called Ty_i *tables*:

$$y = Ty_1(x_1) \oplus Ty_2(x_2) \oplus Ty_3(x_3) \oplus Ty_4(x_4)$$

Definition 2 (Ty_i table)**.** *A Ty_i table maps an 8-bit integer to a 32-bit integer, defined as follows:*

$$Ty_1(x) = x \cdot [\texttt{02}\ \texttt{01}\ \texttt{01}\ \texttt{03}]^\intercal$$
$$Ty_2(x) = x \cdot [\texttt{03}\ \texttt{02}\ \texttt{01}\ \texttt{01}]^\intercal$$
$$Ty_3(x) = x \cdot [\texttt{01}\ \texttt{03}\ \texttt{02}\ \texttt{01}]^\intercal$$
$$Ty_4(x) = x \cdot [\texttt{01}\ \texttt{01}\ \texttt{03}\ \texttt{02}]^\intercal$$

Four copies of each Ty_i table are needed to complete a single `MixColumns` operation. As a result, $16(n_r - 1)$ Ty_i tables are needed for the entire AES encryption function. To save space, the Ty_i tables are composed with the T-boxes. However, the resulting lookup tables only compute the intermediate y_i results. To compute all y values and fully replace `MixColumns`, an additional 12 *XOR tables* are needed, which map two 32-bit integers to a 32-bit integer. In the CEJO framework, these 32-bit XOR tables are implemented by combining eight 4-bit XOR tables.

Definition 3 (XOR table)**.** *An XOR table maps two 4-bit integers to a 4-bit integer, defined as follows:*

$$XOR(x, y) = x \oplus y$$

In total, 96 4-bit XOR tables are needed to complete the `MixColumns` operation, resulting in $96(n_r - 1)$ XOR tables for all rounds.

Figure 3.3 shows a diagram of a single AES round, implemented using lookup tables. For the sake of simplicity, the diagram combines the eight 4-bit XOR tables into a 32-bit XOR table. Note that the `ShiftRows` step is implemented by simply loading the shifted byte $a_{i,j}$ from the state matrix.

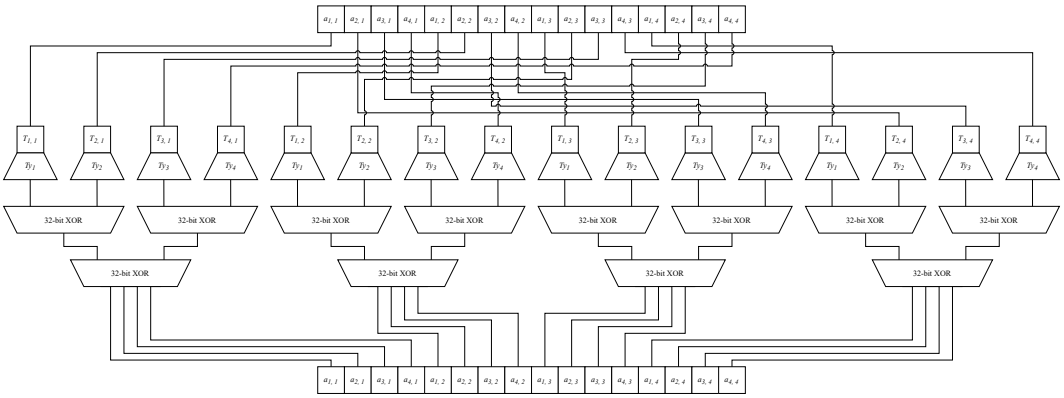

FIGURE 3.3: A single round of AES, implemented using lookup tables

3.2.2 Encodings

Because the table-based implementation of AES is still trivially insecure against attacks in the white-box model, more steps are needed to produce a protected version. In the second step of the CEJO framework, random bijections called *input and output encodings* are applied to every table T.

Definition 4 (Encodings)**.** *Let T be a table. Then $T' = g \circ T \circ f^{-1}$ is an encoded table, where f^{-1} and g are called the input and output encodings, respectively.*

These input and output encodings are applied to the tables in a networked manner. For example, if $T_2 \circ T_1$, then $T_1' = g \circ T_2 \circ f^{-1}$ and $T_2' = h \circ T_2 \circ g^{-1}$. Clearly:

$$T_2' \circ T_1' = (h \circ T_2 \circ g^{-1}) \circ (g \circ T_1 \circ f^{-1}) = h \circ T_2 \circ T_1 \circ f^{-1}$$

This is equivalent to applying the encodings directly to $T_2 \circ T_1$.

Because the input and output encodings are chosen at random, they are expected to be nonlinear. The encodings are then applied at the inputs of the T-boxes, at the outputs of the Ty_i tables, and at all inputs and outputs of the XOR tables. For the full details of the generation and application of these encodings, we refer to [1] and [10].

3.2.3 Mixing bijections

The input and output encodings provide local security to the white-box implementation, that is, it is not feasible to extract information from a single protected table. However, it might still be possible to recover the key when multiple tables from multiple rounds are considered. To prevent this, Chow et al. introduced the concept of *mixing bijections*.

Definition 5 (Mixing bijection)**.** *A mixing bijection is a linear bijection which provides diffusion to disguise the operation of a white-box AES implementation.*

Mixing bijections are applied at the inputs of each T-box and at the outputs of each Ty_i table, except for the T-box of the first round and the Ty_i table of the last round. Similar to the input and output encodings of Section 3.2.2, these mixing bijections are applied in a networked manner. Assuming the 8-bit mixing bijection $K_{i,j}$ has been applied to the byte $a_{i,j}$ from the state matrix, the inverse of this bijection, $K_{i,j}^{-1}$, is composed with the input of the T-box $T_{i,j}^{(r)}$:

$$Ty_i \circ T_{i,j}^{(r)} \circ K_{i,j}^{-1}$$

Applying 32-bit mixing bijections to the outputs of the Ty_i tables is more involved. Suppose Ty_1, Ty_2, Ty_3, and Ty_4 are the Ty_i tables composed with $T_{1,j}^{(r)}$, $T_{2,j}^{(r)}$, $T_{3,j}^{(r)}$, and $T_{4,j}^{(r)}$. To protect the outputs of these tables, a random 32-bit mixing bijection MB_j is generated. MB_j can then be composed with the outputs of Ty_i:

$$MB_j \circ Ty_i \circ T_{i,j}^{(r)} \circ K_{i,j}^{-1}$$

Similar to the implementation of the `MixColumns` operation, the lookup tables required to invert MB_j can be computed by decomposing the multiplication of a 4×4 matrix with a 4×1 vector. In this case, the matrix vector product is applied to each z_i individually, joined by three XOR operations:

$$t = MB_j^{-1} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = MB_j^{-1} \begin{bmatrix} z_1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \oplus MB_j^{-1} \begin{bmatrix} 0 \\ z_2 \\ 0 \\ 0 \end{bmatrix} \oplus MB_j^{-1} \begin{bmatrix} 0 \\ 0 \\ z_3 \\ 0 \end{bmatrix} \oplus MB_j^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ z_4 \end{bmatrix}$$

Once again, lookup tables are used to compute the result of the matrix-vector product. The individual matrix-vector products are replaced by $MB_{i,j}^{-1}$ tables:

$$t = MB_{1,j}^{-1}(z_1) \oplus MB_{2,j}^{-1}(z_2) \oplus MB_{3,j}^{-1}(z_3) \oplus MB_{4,j}^{-1}(z_4)$$

However, before computing the result of the XOR operations, the mixing bijections to protect the T-boxes of the next round must be applied, taking into account the result of the `ShiftRows` operation. Assuming $L_{i,j}$ is the 8-bit mixing bijection to apply to the byte $a_{i,j}$ from the state matrix, this results in the following L_j tables, which are composed with each of the $MB_{i,j}^{-1}$ lookup tables:

$$\begin{aligned} L_1 &= L_{1,1}||L_{2,4}||L_{3,3}||L_{4,2} \\ L_2 &= L_{1,2}||L_{2,1}||L_{3,4}||L_{4,3} \\ L_3 &= L_{1,3}||L_{2,2}||L_{3,1}||L_{4,4} \\ L_3 &= L_{1,4}||L_{2,3}||L_{3,2}||L_{4,1} \end{aligned}$$

In total, 16 additional 8-bit to 32-bit lookup tables and 96 additional 4-bit XOR tables are required to implement the mixing bijections for a single round. Figure 3.4 shows a diagram of a single AES round, protected using mixing bijections. The input and output encodings introduced in Section 3.2.2 are not visible on this diagram.

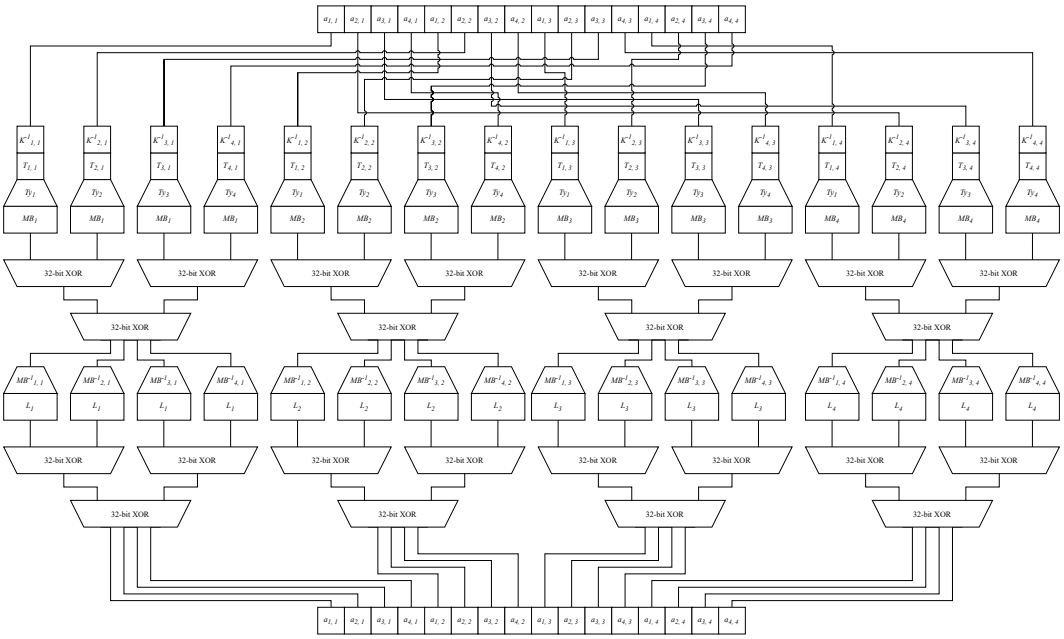

FIGURE 3.4: A single round of AES, protected using mixing bijections

3.2.4 External encodings

Finally, to increase the security of the white-box AES implementation, external encodings are applied to the full encryption function E_k. This ensures that at no

point during encryption or decryption, an unencoded plaintext or ciphertext is visible. In fact, a white-box implementation without external encodings is trivially insecure [1]. Chow et al. recommend to use 128-bit mixing bijections, U^{-1} and V, as external encodings. As a result, each AES encryption round protected using the CEJO framework can be written as a composition of an input encoding, the original round function, and an output encoding.

Definition 6 (Encoded round)**.** *Let $E^{(r)}$ be an AES encryption round. Then the encoded round $\overline{E^{(r)}}$ is defined as follows:*

$$\begin{aligned}\overline{E^{(1)}} &= O^{(1)} \circ E^{(1)} \circ U^{-1} \\ \overline{E^{(r)}} &= O^{(r)} \circ E^{(r)} \circ I^{(r)} \\ \overline{E^{(n_r)}} &= V \circ E^{(n_r)} \circ I^{(n_r)}\end{aligned}$$

with $I^{(r+1)} = (O^{(r)})^{-1}$. Additionally, we also call $I^{(r)}$ and $O^{(r)}$ the internal encodings of round $E^{(r)}$.

Because the internal encodings for a single encryption round are cancelled out, a full AES encryption function protected using the CEJO framework, denoted with $\overline{E_k}$, can be written as follows:

$$\begin{aligned}\overline{E_k} &= \overline{E^{(n_r)}} \circ \cdots \circ \overline{E^{(1)}} \\ &= V \circ E^{(n_r)} \circ I^{(n_r)} \circ \cdots \circ O^{(1)} \circ E^{(1)} \circ U^{-1} \\ &= V \circ E^{(n_r)} \circ \cdots \circ E^{(1)} \circ U^{-1}\end{aligned}$$

Although essential for the security of white-box implementations, external encodings also represent its main disadvantage. A white-box encryption function using external encodings will map the plaintext m to the ciphertext $c' = (V \circ E_k \circ U^{-1})(m)$, as opposed to the ciphertext $c = E_k(m)$ for an unprotected encryption function. Because of this, white-box cryptography might not be possible in situations where cryptographic standards, such as the AES FIPS 197 [11], must be strictly followed.

3.3 Further developments

Shortly after publishing their work on AES, Chow et al. also applied their method to the protection of the DES block cipher [12]. Concurrently, a practical side-channel attack on the white-box DES implementation was published by Jacob et al., using Differential Fault Analysis [13]. However, this attack was not applicable to the CEJO AES implementation. Still, it would take only two years for the initial AES implementation to be broken. In 2004, Billet et al. designed a practical key recovery attack by analyzing the composition of the AES lookup tables [14].

The publication of these papers sparked more interest in the topic of white-box cryptography, with new constructions based on DES [15] and AES [16] [17] [18] [19] [20] [21] appearing over the years. This also spurred research into entirely

different types of constructions using modified cipher designs [22] or incompressible white-box ciphers [23] [24]. Unfortunately, all block cipher implementations based on the fixed-key approach have been broken, using both algebraic attacks [25] [26] [27] [28] [29] [30] [31] [32] [33] and attacks based on side-channel analysis [34] [35] [36]. With the exception of [22], [23], and [24], all of these designs improved upon or were inspired by the CEJO framework. Consequently, this framework has been analyzed extensively.

In 2016, McMillion et al. used a type of permutations called *self-equivalences* to construct a toy white-box implementation of AES [4]. In the same work, they also presented a practical attack recover the cryptographic key from such implementations. Their work received little attention, and was only recently picked up by Ranea et al. [5]. Ranea et al. analyzed the white-box security of substitution-permutation network (SPN) ciphers protected using self-equivalence encodings. They proposed a generic attack on such implementations, and proved that it is possible to recover the key of traditional SPN ciphers if the S-box do not have differential and linear approximations with probability one. As cryptographically strong S-boxes are designed to resist differential [37] and linear [38] cryptanalysis, they showed that self-equivalence encodings are unsuitable to protect this class of traditional SPN ciphers. On the other hand, they also indicated that self-equivalence encodings might be of interest to protect ciphers with a better self-equivalence structure.

Most research into white-box cryptography of block ciphers has been applied to the AES and DES ciphers. As a result, round encoding techniques are mainly developed for block ciphers employing S-boxes, used by AES and DES [9] [39]. However, more modern ciphers are also based on the add-rotate-xor (ARX) operations, whose rounds consist of the three basic operations the name implies: modular addition, bitwise rotation, and bitwise XOR. Curiously, there does not seem to be any previous academic research on applying white-box cryptography to ARX ciphers. One possible reason is that, as described previously, the method introduced by Chow et al. uses lookup tables to generate a white-box AES or DES implementation. For ARX ciphers, it is not feasible to generate a lookup table implementation, as the modular addition function is too big.

Because ARX ciphers do not rely on cryptographically strong S-boxes to provide nonlinearity, they are not susceptible to the attack described by Ranea et al. Furthermore, in [40], it was found that the n-bit modular addition has an exponential number of self-equivalences in n. As a result, ciphers employing the modular addition as their only source of nonlinearity are a promising target for future white-box cryptography research based on self-equivalence encodings. In this thesis, we aim to introduce ARX ciphers to white-box cryptography, and encourage further academic research in this direction.

Some examples of modern, well-known ARX ciphers include Salsa20, a family of 256-bit stream ciphers designed by Daniel J. Bernstein in 2008 [41], ChaCha, a variant of Salsa20 also designed by Bernstein [42], Threefish (2010) [43], and Speck, a collection of block ciphers published by the NSA in 2013 [6]. Of these possible ARX ciphers, Speck employs by far the simplest round function. The Salsa20 and ChaCha quarter-round functions each consist of four modular

addition functions, resulting in sixteen modular additions for a single round. Similarly, Threefish contains two modular additions per round. On the other hand, Speck only requires one modular addition function in its round function. Furthermore, Salsa20 and ChaCha are stream ciphers, whereas Speck has a block cipher structure. This makes it easier for us to build on the previous work on self-equivalence encodings by Ranea et al.

Chapter 4

Self-equivalences

In this chapter, we introduce the mathematical concept of *self-equivalences*, permutations A and B such that $F = B \circ F \circ A$. After introducing self-equivalences, we look at how these permutations can be used to protect the round keys of SPECK implementations. Finally, the generation of random linear and affine self-equivalences of the SPECK modular addition is discussed in Section 4.3 and Section 4.4 of this chapter.

4.1 Definitions

In this section we introduce the definitions of linear and affine self-equivalences.

Definition 7 (Linear self-equivalences)**.** *Let F be an (n, m)-bit function. Let A be an n-bit linear permutation and B be an m-bit linear permutation. If $F = B \circ F \circ A$, we call the pair (A, B) a linear self-equivalence of F.*

Because A and B are linear functions, they could be given in the form of $n \times n$ and $m \times m$ matrices, respectively. In that case, we say (A, B) is a linear self-equivalence of F in matrix form.

Definition 8 (Affine self-equivalences)**.** *Let F be an (n, m)-bit function. Let A be an n-bit linear permutation, a an n-bit constant, B be an m-bit linear permutation, and b an m-bit constant. Together, (A, a) and (B, b) describe affine permutations. If $F = (\oplus_b \circ B) \circ F \circ (\oplus_a \circ A)$, we call the pair $((A, a), (B, b))$ an affine self-equivalence of F, or just a self-equivalence of F.*

Similarly, A, a, B, and b could be given in the form of $n \times n$ and $m \times m$ matrices, and vectors of length n and m, respectively. In that case, we say $((A, a), (B, b))$ is an (affine) self-equivalence of F in matrix-vector form. Of course, linear self-equivalences are also affine self-equivalences, with a and b equal to 0.

During this thesis, we will mostly work with the matrix and matrix-vector forms of self-equivalences. This allows us to precisely specify the self-equivalences we are using, as well as manipulate these matrices and vectors using basic linear algebra.

4.2 Self-equivalences and Speck

In Section 2.2, we introduced the Feistel-like structure and encryption round function of Speck. We briefly repeat the round function in Algorithm 2.

Algorithm 2 Speck encryption round function

$x, y \leftarrow$ encryption round input
$x \leftarrow x \ggg \alpha$
$x \leftarrow x \boxplus y$
$x \leftarrow x \oplus k^{(r)}$
$y \leftarrow y \lll \beta$
$y \leftarrow x \oplus y$
return x, y

A schematic representation of the Speck round function is shown in Figure 4.1.

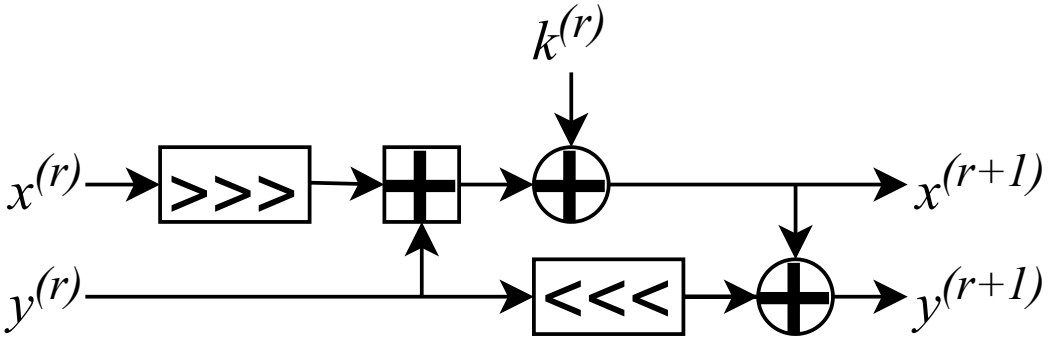

Figure 4.1: Diagram of a Speck encryption round

To protect the round keys $k^{(r)}$ from key recovery attacks using self-equivalences, we need to rewrite the Speck encryption function as a substitution-permutation network (SPN). We start by defining the encryption function of an SPN.

Definition 9 (SPN encryption function)**.** *Let E_k be an encryption function which takes a plaintext m and encrypts this plaintext using key k to produce ciphertext c. Then E_k represents the encryption function of a substitution-permutation network if E_k can be decomposed in affine layers AL and substitution layers SL as follows:*

$$E_k = AL^{(n_r)} \circ SL \circ \cdots \circ AL^{(2)} \circ SL \circ AL^{(1)}$$

In addition, we call $SL \circ AL^{(r)}$ an SPN encryption round $E^{(r)}$.

We can now prove that the Speck encryption function can also be written as a combination of SPN encryption rounds.

Theorem 1. *Let E_k be the encryption function of the SPECK cipher consisting of n_r rounds with word size n. E_k can be decomposed in affine layers AL and substitution layers SL, $E_k = AL^{(n_r)} \circ SL \circ \cdots \circ AL^{(1)} \circ SL \circ AL^{(0)}$, with:*

$$\begin{aligned} SL(x, y) &= (x \boxplus y, y) \\ AL^{(0)}(x, y) &= (x \ggg \alpha, y) \\ AL^{(r)}(x, y) &= ((x \oplus k^{(r)}) \ggg \alpha, (x \oplus k^{(r)}) \oplus (y \lll \beta)), \text{ for } 1 \leq r \leq n_r - 1 \\ AL^{(n_r)}(x, y) &= (x \oplus k^{(n_r)}, (x \oplus k^{(n_r)}) \oplus (y \lll \beta)) \end{aligned}$$

We prove this theorem by illustrating the transformations visually through Figure 4.2. Here, two SPECK rounds are shown in sequence, with the dotted lines indicating the affine layers separated by modular additions. Evidently, this can be extended to n_r SPECK rounds, resulting in $n_r + 1$ affine layers, where layer 0 and n_r have a special structure.

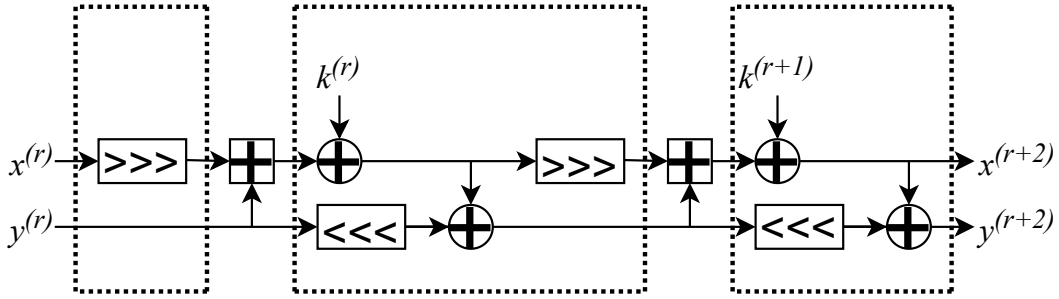

FIGURE 4.2: Diagram of two SPECK encryption rounds, with affine layers indicated using dotted lines

In the previous definitions of AL, the SPECK state consists of two n-bit variables x and y. However, the self-equivalences of SL are $2n$-bit affine permutations, which operate on vectors of length $2n$ with elements in $\mathbb{F}_2$. To be able to apply these self-equivalences to AL, we need rewrite AL as $2n$-bit affine permutations operating on a $2n$-bit state vector xy. Here, xy contains the bits of x and y in little-endian order.

Definition 10 (AL as $2n$-bit permutations)**.**

$$\begin{aligned} AL^{(0)} &= R_\alpha \\ AL^{(r)} &= R_\alpha \circ X \circ L_\beta \circ \oplus_{k'^{(r)}}, \text{ for } 1 \leq r \leq n_r - 1 \\ AL^{(n_r)} &= X \circ L_\beta \circ \oplus_{k'^{(n_r)}} \end{aligned}$$

where R_α represents a right bitwise rotation of x by α positions, L_β represents a left bitwise rotation of y by β positions, and X represents the bitwise XOR operation such that $y = x \oplus y$. Finally, $k'^{(r)}$ is a vector of length $2n$ containing the key bits of the round key $k^{(r)}$ in the first n positions and zero in the last n positions.

Figure 4.3 shows two Speck SPN encryption rounds in terms of AL and modular additions. We will omit the details on adapting the modular addition to a state vector xy, as this change mostly depends on the implementation, discussed in Chapter 6.

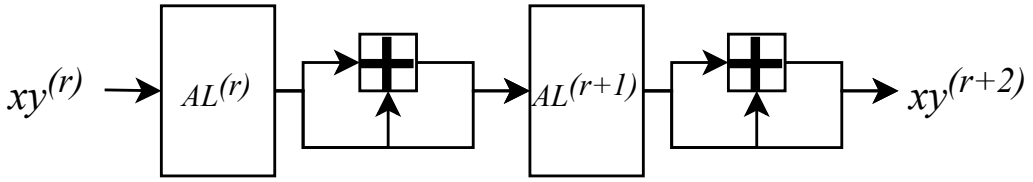

Figure 4.3: Diagram of two Speck SPN encryption rounds, expressed using affine layers and substitution layers

Now that $AL^{(r)}$ are $2n$-bit affine permutations, we can introduce the definition of an *encoded affine layer* $\overline{AL^{(r)}}$.

Definition 11 (Encoded affine layer)**.** *Let $AL^{(r)}$ be an affine layer of the Speck cipher, with $1 \leq r \leq n_r$. Let $((O^{(r)}, o^{(r)}), (I^{(r+1)}, i^{(r+1)}))$ be a self-equivalence of the Speck substitution layer SL. Then we call $\overline{AL^{(r)}}$ an encoded affine layer, with:*

$$\overline{AL^{(r)}} = (\oplus_{o^{(r)}} \circ O^{(r)}) \circ AL^{(r)} \circ (\oplus_{i^{(r)}} \circ I^{(r)})$$

In addition, we call $(I^{(r)}, i^{(r)})$ and $(O^{(r)}, o^{(r)})$ the self-equivalence encodings of AL.

Note that $AL^{(0)}$ will not be encoded using self-equivalences: this affine layer does not contain any key material, so it can be skipped.

If the self-equivalences composed with each $AL^{(r)}$ are sampled randomly from a set of self-equivalences, the unencoded affine layer $AL^{(r)}$ can not be recovered without knowledge of $(I^{(r)}, i^{(r)})$ and $(O^{(r)}, o^{(r)})$. This effectively hides the round keys inside the affine layers, and is the basis of our method to protect Speck implementations using self-equivalence encodings. Moreover, this process could easily be adapted to other ARX ciphers, as long as there is a method to generate random self-equivalences of the substitution layer.

We will describe methods to generate random linear and affine self-equivalences in the following sections. However, we first need to introduce the *external encodings*, which are essential to the security of white-box cryptographic implementations.

Definition 12 (External encodings)**.** *Let $\overline{AL^{(1)}}$ and $\overline{AL^{(n_r)}}$ be the first, respectively last, encoded affine layers of a Speck implementation with n_r rounds. Then $(\oplus_{i^{(1)}} \circ I^{(1)})$ and $(\oplus_{o^{(n_r)}} \circ O^{(n_r)})$ are random affine permutations, called the input and output encoding, respectively. Together, we call these permutations the external encodings.*

Definition 13 (Encoded encryption function)**.** *Let E_k be the encryption function of the Speck cipher consisting of n_r rounds with word size n. We call $\overline{E_k}$ an encoded Speck encryption function, with:*

$$\overline{E_k} = \overline{AL^{(n_r)}} \circ SL \circ \cdots \circ \overline{AL^{(1)}} \circ SL \circ AL^{(0)}$$

Additionally, we call $SL \circ \overline{AL^{(r)}}$ an encoded round $\overline{E^{(r)}}$.

By applying the property of self-equivalences to $\overline{AL^{(r)}}$ of an encoded Speck encryption function $\overline{E_k}$, we obtain:

$$\begin{aligned}\overline{E_k} &= \overline{AL^{(n_r)}} \circ SL \circ \cdots \circ \overline{AL^{(1)}} \circ SL \circ AL^{(0)} \\ &= (\oplus_{o^{(n_r)}} \circ O^{(n_r)}) \circ AL^{(n_r)} \circ SL \circ \cdots \circ AL^{(1)} \circ (\oplus_{i^{(1)}} \circ I^{(1)}) \circ SL \circ AL^{(0)} \\ &= (\oplus_{o^{(n_r)}} \circ O^{(n_r)}) \circ AL^{(n_r)} \circ SL \circ \cdots \circ AL^{(1)} \circ SL \circ AL^{(0)} \circ (\oplus_{i'^{(1)}} \circ I'^{(1)}) \\ &= (\oplus_{o^{(n_r)}} \circ O^{(n_r)}) \circ E_k \circ (\oplus_{i'^{(1)}} \circ I'^{(1)})\end{aligned}$$

Clearly, applying self-equivalence encodings to the affine layers does not change the behavior of $\overline{E_k}$, provided that the inputs and outputs are encoded using the external encodings. This property is also illustrated on Figure 4.4, for two encryption rounds. The dotted lines indicate the substitution layer SL surrounded by its self-equivalence, which can simply be reduced to SL.

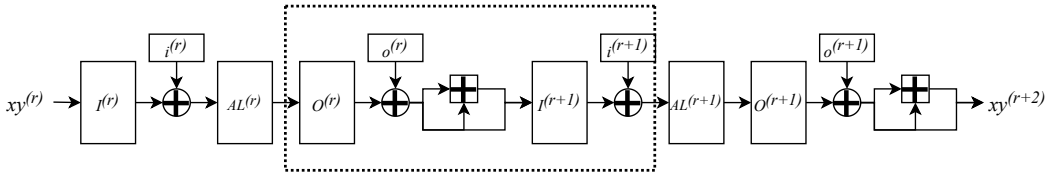

Figure 4.4: Diagram of two Speck SPN encryption rounds encoded using self-equivalences

As an example, suppose Alice wants to securely receive a message m from Bob, and Bob has access to an encoded encryption function $\overline{E_k}$. Bob starts by using $\overline{E_k}$ to encrypt m:

$$c = \overline{E_k}(m)$$

Alice can then use her knowledge of the key k and the external encodings to decrypt c by computing:

$$\begin{aligned}m &= ((\oplus_{i'^{(1)}} \circ I'^{(1)})^{-1} \circ D_k \circ (\oplus_{o^{(n_r)}} \circ O^{(n_r)})^{-1})(c) \\ &= \overline{E_k}^{-1}(\overline{E_k}(m))\end{aligned}$$

4.3 Generating linear self-equivalences

In this section, we introduce a method to generate linear self-equivalences for the Speck substitution layer SL. This method was first described by Ranea in [40]. Given any word size n, we start by introducing $n \times n$ matrices A_i and B_i:

$$A_i = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \vdots & 0 & \ddots & 0 & \vdots \\ 0 & 0 & 0 & 1 & 0 \\ a_{i,1} & a_{i,2} & \dots & a_{i,n-1} & 1 \end{bmatrix}, B_i = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \vdots & 0 & \ddots & 0 & \vdots \\ 0 & 0 & 0 & 0 & 0 \\ b_{i,1} & b_{i,2} & \dots & b_{i,n-1} & 0 \end{bmatrix}$$

with $b_{0,j} = a_{0,j}$ and $b_{1,j} = a_{0,j} + a_{1,j}$ for $2 \leq j \leq n-1$.

These matrices can be composed to construct the set $\mathcal{A}$ of $4n$-bit linear functions, represented as a block matrix:

$$\mathcal{A} = \begin{bmatrix} A_0 & B_0 & B_0 & 0 \\ B_1 & A_1 & A_0 + A_1 & B_0 \\ B_0 & 0 & A_0 & B_0 \\ A_0 + A_1 & B_0 & B_1 & A_1 \end{bmatrix}$$

Because there are $2n$ free coefficients ($a_{0,1}, \ldots, a_{0,n-1}$, $a_{1,1}, \ldots, a_{1,n-1}$, $b_{0,1}$, and $b_{1,1}$) in $\mathbb{F}_2$, $|\mathcal{A}| = 2^{2n}$.

Finally, we need to introduce the $4n$-bit linear permutation L, defined by the following block matrix:

$$L = \begin{bmatrix} 0 & I_n & I_n & 0 \\ I_n & I_n & I_n & 0 \\ 0 & 0 & I_n & 0 \\ I_n & 0 & I_n & I_n \end{bmatrix}$$

Using $\mathcal{A}$ and L, we can generate 2^{2n} different linear self-equivalences of the modular addition. Theorem 2, adapted from Proposition 1 from [40], describes the final step in the generation method.

Theorem 2 (Proposition 1 from [40])**.** *Let $\widehat{\mathcal{A}}$ be the set of $4n$-bit linear functions given by $L \circ \mathcal{A} \circ L^{-1}$. Then, any function $F \in \widehat{\mathcal{A}}$ is of the form $F(x, y, x', y') = M_1(x, y)||M_2(x', y')$ and (M_1, M_2^{-1}) is a linear self-equivalence of SL, that is, $M_2 \circ SL = SL \circ M_1$. In particular, the modular addition has at least 2^{2n} linear self-equivalences.*

We call the set of linear self-equivalences generated using this method $SE_L(SL)$, that is, $SE_L(SL) = \widehat{\mathcal{A}}$. As mentioned previously, $|SE_L(SL)| = 2^{2n}$. This is important for the security of our method to protect Speck implementations: the number of self-equivalences should be as high as possible to prevent a simple brute-force key recovery attack. For $n = 64$, the largest Speck word size, this would result in 2^{128} possibilities, enough to resist a naive brute-force attack. An extensive security analysis of linear self-equivalence encodings can be found in Chapter 5.

4.4 Generating affine self-equivalences

In [40], Ranea also includes an algorithm to generate two different sets of of affine self-equivalences for the Speck modular addition function SL. We will call these sets the *type 1* and *type 2* affine self-equivalences.

Similar to the linear self-equivalences, we start by introducing the sets $\mathcal{A}_1$ and $\mathcal{A}_2$, containing $4n$-bit affine functions (with n the Speck word size). These sets are used to generate type 1 and type 2 affine self-equivalences, respectively. By definition, $|\mathcal{A}_1| = 2^{2n+7}$ and $|\mathcal{A}_2| = 3 \times 2^{2n+5}$. However, the exact definitions of these sets are too long to be included in this section. We refer to the Python implementation part of this thesis for more information on generating $\mathcal{A}_1$ and $\mathcal{A}_2$.

Now, we also define the $4n$-bit linear permutations L_1 and L_2, defined by the block matrices:

$$L_1 = \begin{bmatrix} I_n & 0 & I_n & I_n \\ 0 & 0 & I_n & I_n \\ 0 & 0 & I_n & 0 \\ 0 & I_n & I_n & 0 \end{bmatrix}, L_2 = \begin{bmatrix} I_n & 0 & I_n & I_n \\ 0 & I_n & I_n & 0 \\ 0 & 0 & I_n & 0 \\ 0 & 0 & I_n & I_n \end{bmatrix}$$

Theorem 3 describes the method to generate type 1 and type 2 affine self-equivalences using $\mathcal{A}_1$, L_1, $\mathcal{A}_2$, and L_2. Note that, instead of writing $((A, a), (B, b))$ to indicate an affine self-equivalence, we simply use the notation (A, B), with A and B affine permutations.

Theorem 3 (Proposition 2 from [40])**.** *Let $\widehat{\mathcal{A}_i}$ be the set of $4n$-bit linear functions given by $L_i \circ \mathcal{A}_i \circ L_i^{-1}$, where $i \in \{1, 2\}$. Then, any function $F \in \widehat{\mathcal{A}_i}$ is of the form $F(x, y, x', y') = M_1(x, y)||M_2(x', y')$ and (M_1, M_2^{-1}) is an affine self-equivalence of SL, that is, $M_2 \circ SL = SL \circ M_1$. Moreover, $\widehat{\mathcal{A}_1} \circ \widehat{\mathcal{A}_2} \supsetneq \widehat{\mathcal{A}_1} \cup \widehat{\mathcal{A}_2}$, thus $|\widehat{\mathcal{A}_1} \circ \widehat{\mathcal{A}_2}| > max(|\widehat{\mathcal{A}_1}|, |\widehat{\mathcal{A}_2}|)$. In particular, the modular addition has more than 2^{2n+7} affine self-equivalences.*

The set of type 1 affine self-equivalences of SL is called $SE_1(SL) = \widehat{\mathcal{A}_1}$ and the set of type 2 affine self-equivalences of SL is called $SE_2(SL) = \widehat{\mathcal{A}_2}$. Because the affine self-equivalences of SL form a group, they can easily be composed to construct new affine self-equivalences. In the interest of maximizing the self-equivalence search space, we would like to generate affine self-equivalences from $SE_1(SL) \circ SE_2(SL)$. Theorem 3 also states that combining $SE_1(SL)$ and $SE_2(SL)$ results in more affine self-equivalences, although a specific bound is not given. Consequently, when generating a random affine self-equivalence to protect a SPECK implementation, we generate one affine self-equivalence each from $SE_1(SL)$ and $SE_2(SL)$, and compose these self-equivalences to obtain a new affine self-equivalence. For a word size $n = 64$, this results in more than 2^{135} possibilities, more than sufficient to prevent a simple brute-force attack.

Chapter 5

Security analysis of white-box Speck

In this chapter, we analyze the security of our white-box Speck method. We start by giving a short introduction to the attack model, attacker goals, and common attacks on white-box implementations. In Section 5.2, we discuss the inversion of the Speck key schedule, used to reconstruct the master key from round keys. In the final two sections, Section 5.3 and Section 5.4, we take a closer look at the security of implementations using linear and affine encodings, respectively.

5.1 Introduction

The security of white-box implementations can be expressed in many different ways. Most commonly, the goal of the attacker is to extract the cryptographic key from a provided implementation (*key extraction*). However, other security notions include *one-wayness* and *incompressibility*. In the case of one-wayness, an attacker should be unable to invert the functionality of the cipher, effectively transforming a secret-key scheme into a public-key scheme. Incompressibility refers to the size of the white-box implementation: an implementation is incompressible if the functionality of the cipher is lost when parts of the implementation are removed or compressed. This impedes the redistribution of white-box implementations with a very large size. A detailed analysis of white-box cryptography security goals can be found in [44]. In this thesis, we focus on the fundamental white-box security feature: resistance to key recovery attacks.

In white-box cryptography, an attacker is much more powerful compared to conventional cryptography. Recall that the white-box model has three main properties (Section 3.1):

- The attacker is a privileged user on the same host as the cryptographic algorithm, with complete access to the implementation.
- The attacker can dynamically execute the cryptographic algorithm.

- At any point before, during, or after the execution, the attacker is able to view and modify the internal details of the implementation.

There are many different approaches to exploit these capabilities in attacks on white-box implementations. For example, one could try to attack the CEJO framework by reducing the search space of possible round encodings. This would enable a brute-force search on the encoded round to recover the round key [14]. Other popular techniques are based on side-channel analysis, such as *differential fault analysis* (DFA) [45] and *differential computation analysis* (DCA) [34].

In our analysis, we will evaluate the security of our white-box Speck method from an algebraic perspective. Although self-equivalence encodings are generated at random, they are not completely random linear or affine transformations. We will try to exploit this additional structure to reduce the brute-force search space of possible self-equivalence encodings and recover key bits. Moreover, to fully compromise the security, we will also need to recover the external encodings from the white-box implementation. In the broader context of the white-box model, our approach is quite simple: we only require access to the encoded affine layers of the implementation.

5.2 Speck key schedule inversion

Traditionally, block ciphers employ a key schedule to transform a user-provided master key, k, into multiple round keys, $k^{(r)}$. An important goal of a key schedule is to minimize the relations between these round keys, to prevent cryptographic attacks such as slide attacks. As introduced in Section 2.2, Speck uses its own round function to generate $k^{(r)}$ from k. We briefly repeat the description of the key schedule here. Let n be the Speck word size, and m the number of key words, that is, the key size divided by n [6]. Then k can be written as a sequence of m key words ($2 \leq m \leq 4$):

$$k = \begin{bmatrix} l^{(m-1)} & \dots & l^{(1)} & k^{(1)} \end{bmatrix}$$

Now, $l^{(r+m-1)}$ and $k^{(r+1)}$ can be computed by applying the Speck round function:

$$l^{(r+m-1)} = (k^{(r)} \boxplus (l^{(r)} \ggg \alpha)) \oplus r$$
$$k^{(r+1)} = (k^{(r)} \lll \beta) \oplus l^{(r+m-1)}$$

To perform a key recovery attack on the white-box Speck implementation, we need to recover the master key k from the encoded implementation $\overline{E_k}$. Unfortunately, the encoded implementation only contains protected versions of the round keys, $k^{(r)}$. As a result, recovering k directly is not possible, so computing k using some recovered $k^{(r)}$ is a crucial part of a successful key recovery attack. Luckily, the Speck key schedule is invertible, and k can be computed easily, using only the m first round keys. Suppose $k^{(1)}, \dots, k^{(m)}$ are known, then compute:

$$l^{(r+m-1)} = (k^{(r)} \lll \beta) \oplus k^{(r+1)}$$
$$l^{(r)} = ((l^{(r+m-1)} \oplus r) \boxminus k^{(r)}) \lll \alpha$$

Combining $l^{(m-1)}, \dots, l^{(1)}$, and $k^{(1)}$, we obtain the master key k.

Note that this approach could be extended to reconstruct k using any sequence of m consecutive round keys. However, as the security analysis introduced in this chapter is applicable to any encoded round, we will always choose to attack the m first encoded rounds.

5.3 Security analysis of linear self-equivalences

We start the analysis of the white-box method for SPECK by looking at a variant where all encodings, both self-equivalence encodings and external encodings, are linear. Although linear encodings are significantly weaker than affine encodings in terms of security, they are also conceptually easier to understand. Furthermore, the analysis of this weaker version might give us some initial insights in the security of a more secure variant using affine encodings.

In this section, we will focus on a single affine layer of an encoded SPECK encryption function $\overline{E_k}$. For the sake of convenience, we repeat the definition of an encoded affine layer for round r (see Definition 11) here:

$$\overline{AL^{(r)}} = (\oplus_{o^{(r)}} \circ O^{(r)}) \circ AL^{(r)} \circ (\oplus_{i^{(r)}} \circ I^{(r)})$$

Because we only consider linear encodings in this section, $i^{(r)}$ and $o^{(r)}$ are zero vectors. As a result, the definition can be simplified to:

$$\overline{AL^{(r)}} = O^{(r)} \circ AL^{(r)} \circ I^{(r)}$$

Generally, this encoded affine layer will be stored as a combination of an encoded matrix $\overline{M^{(r)}}$ and an encoded vector $\overline{v^{(r)}}$. For each round r, $M^{(r)}$ represents the known linear operations of the affine layer, while $v^{(r)}$ is the constant of the affine layer. However, as $v^{(0)}$ does not contain any key material, this round is not protected using self-equivalences and we will not consider it in our analysis:

$$\begin{aligned}
\overline{M^{(0)}} &= M^{(0)} \\
\overline{v^{(0)}} &= v^{(0)} \\
\overline{M^{(r)}} &= O^{(r)} M^{(r)} I^{(r)}, \text{ for } 1 \leq r \leq n_r \\
\overline{v^{(r)}} &= O^{(r)} v^{(r)}, \text{ for } 1 \leq r \leq n_r
\end{aligned}$$

Armed with these definitions, we identify two main approaches to attempt a key recovery attack on our white-box SPECK method.

Firstly, we might try to recover key bits from $\overline{v^{(r)}}$ directly. While $O^{(r)}$ is generated at random from the set of possible self-equivalences of SL, there is no guarantee that every entry in the matrix is random too. Consequently, the matrix-vector product might not hide all elements of $v^{(r)}$. This could lead to some key bits being recovered. However, it is not possible to obtain the external encodings using this approach.

Secondly, we could attempt to recover $I^{(r)}$ or $O^{(r)}$ from $\overline{M^{(r)}}$ or $\overline{v^{(r)}}$. If any of these self-equivalence encodings could be computed, the security of the round is completely compromised and all the key bits could be recovered easily. Additionally, external encodings $I^{(1)}$ and $O^{(n_r)}$ could be computed from $\overline{M^{(1)}}$ and $\overline{M^{(n_r)}}$ if $O^{(1)}$ and $I^{(n_r)}$ are known.

5.3.1 Recovering key bits from $\overline{v^{(r)}}$

In Section 4.3, we introduced a method to generate linear self-equivalences of the modular addition. Given a word size n and $2n$ free coefficients, 2^{2n} different linear self-equivalences can be generated using this method. Let $c_i^{(r)}$, with $1 \leq i \leq 2n$, be the coefficients used to generate the linear self-equivalence $(O^{(r)}, I^{(r+1)})$. These coefficients should be generated at random and are unknown to the attacker, similar to a random seed. The shape of $O^{(r)}$ and $I^{(r+1)}$ as $2n \times 2n$ matrices is given by:

$$\left[\begin{array}{cccccc|cccccc}
1 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\
\vdots & 0 & \ddots & 0 & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\
0 & 0 & \dots & 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\
0 & 0 & \dots & 0 & 0 & 1 & c & c & \dots & c & c & c \\
\hline
0 & 0 & \dots & 0 & 0 & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\
0 & 0 & \dots & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & 0 & \ddots & 0 & \vdots & \vdots \\
0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 \\
c & 0 & \dots & 0 & 0 & 0 & c & c & \dots & c & c & 1
\end{array}\right]$$

Here, c denotes a symbolic value which depends on one or more coefficients $c_i^{(r)}$. Furthermore, each $c_i^{(r)}$ is present at least once in both $O^{(r)}$ and $I^{(r+1)}$.

Clearly these linear transformations are far from random: only $2n$ of the $4n^2$ entries in the matrix actually depend on the coefficients used to generate the self-equivalences. On top of this, the matrices are also very sparse, containing only $4n$ nonzero entries. These properties are disastrous for the security of linear self-equivalence encodings. Indeed, computing the matrix-vector product in a symbolic way, $\overline{v^{(r)}}$ simplifies to:

$$\begin{aligned}
\overline{v^{(r)}} &= O^{(r)} v^{(r)} \\
&= \left[\begin{array}{cccc|ccc}
v_1^{(r)} & \dots & v_{n-1}^{(r)} & \overline{v^{(r)}}_n & v_{n+1}^{(r)} & \dots & v_{2n-1}^{(r)} & \overline{v^{(r)}}_{2n}
\end{array}\right]
\end{aligned}$$

Only two of the $2n$ elements of $v^{(r)}$, and one of the n bits of $k^{(r)}$, are protected by the secret coefficients $c_i^{(r)}$. This allows the attacker to compute candidate round keys by only guessing the value of a single bit. Candidate master keys could simply be enumerated by guessing 2^m values (in the worst case) for m consecutive round keys, compared to a theoretical 2^{mn} guesses required without additional information (where m depends on the original key size).

Fortunately, this approach does not reveal the external encodings $I^{(1)}$ and $O^{(n_r)}$ to the attacker. As the attacker is unable to use or verify a candidate master key without the external encodings, the impact of this vulnerability is low if external encodings are used and randomly generated. However, this vulnerability does show

that the white-box SPECK variant using only linear self-equivalence encodings is trivially insecure, if external encodings are not used.

5.3.2 Recovering encodings

Recall that $\overline{M^{(r)}}$ is computed as the matrix product of $O^{(r)}$, $M^{(r)}$, and $I^{(r)}$. Consequently, $\overline{M^{(r)}}$ contains all coefficients used to generate $I^{(r)}$ and $O^{(r)}$, that is, $c_i^{(r-1)}$ and $c_i^{(r)}$ with $1 \leq i \leq 2n$. To ensure the security of the white-box SPECK implementation, it is crucial that these coefficients are not easily extracted from $\overline{M^{(r)}}$. After all, if these coefficients could be recovered, an attacker could simply reuse them to compute the self-equivalences and recover the round keys.

Unfortunately, we know that $I^{(r)}$ and $O^{(r)}$ are very sparse matrices. When the matrix product of $O^{(r)}M^{(r)}$ and $I^{(r)}$ is computed, the many zeroes in $I^{(r)}$ will partially erase the intermediate results, and leave the coefficients of $I^{(r)}$ open to extraction. This weakness will be exploited in our second approach to recover the external encodings and round keys of a white-box SPECK implementation.

We first describe an algorithm to recover the coefficients $c_i^{(r-1)}$ from $\overline{M^{(r)}}$ (Algorithm 3). These are the coefficients used to generate the self-equivalence $(O^{(r-1)}, I^{(r)})$. Notably, this algorithm does not involve any brute-force search: many coefficients are trivially extracted from $\overline{M^{(r)}}$, and only limited computation is necessary to obtain the rest.

Algorithm 3 Recovering $c_i^{(r-1)}$ from $\overline{M^{(r)}}$

$j \leftarrow 1$
for $j < n - 1$ **do**
  $c_{2n-j}^{(r-1)} \leftarrow \overline{M^{(r)}}_{n-\alpha,n+1+j}$
  $c_{n+1-j}^{(r-1)} \leftarrow \overline{M^{(r)}}_{n+\beta,n+1+j} + c_{2n-j}^{(r-1)}$
  $j \leftarrow j + 1$
end for
$c_2^{(r-1)} \leftarrow \overline{M^{(r)}}_{n-\alpha,n+1}$
$c_{n+1}^{(r-1)} \leftarrow \overline{M^{(r)}}_{n+\beta,n+1} + c_2^{(r-1)}$
$c_{2n}^{(r-1)} \leftarrow \overline{M^{(r)}}_{n-\alpha,1} + c_2^{(r-1)}$
$c_1^{(r-1)} \leftarrow \overline{M^{(r)}}_{n+\beta,1} + c_{n+1}^{(r-1)} + c_{2n}^{(r-1)}$
return $c_i^{(r-1)}$

By executing Algorithm 3 on $\overline{M^{(2)}}$, an attacker can recover $c_i^{(1)}$ and generate the self-equivalence $(O^{(1)}, I^{(2)})$ using the method described in Section 4.3. Because $M^{(1)}$ is always publicly known, the attacker can compute

$$\begin{aligned}(O^{(1)}M^{(1)})^{-1}\overline{M^{(1)}} &= (O^{(1)}M^{(1)})^{-1}O^{(1)}M^{(1)}I^{(1)} \\ &= I^{(1)}\end{aligned}$$

to obtain the input external encoding $I^{(1)}$.

Recovering the output external encoding, $O^{(n_r)}$, requires more effort. As $\overline{O^{(n_r)}}$ is generated uniformly at random, it is not possible to apply Algorithm 3 to $\overline{M^{(n_r)}}$ and recover $c^{(n_r-1)}$. One possible alternative approach is to try to recover $c^{(n_r-1)}$ from $\overline{M^{(n_r-1)}}$, use this to generate the self-equivalence $(O^{(n_r-1)}, I^{(n_r)})$, and obtain $O^{(n_r)}$ analogous to $I^{(1)}$. However, we found that the coefficients $c^{(r)}$ are significantly harder to extract from $\overline{M^{(r)}}$ compared to $c^{(r-1)}$.

Instead, the attacker starts by executing Algorithm 3 on $\overline{M^{(n_r-1)}}$ to recover $(O^{(n_r-2)}, I^{(n_r-1)})$. Then, the attacker computes

$$\begin{aligned}\overline{M^{(n_r-1)}}(M^{(n_r-1)}I^{(n_r-1)})^{-1} &= O^{(n_r-1)}M^{(n_r-1)}I^{(n_r-1)}(M^{(n_r-1)}I^{(n_r-1)})^{-1} \\ &= O^{(n_r-1)}\end{aligned}$$

to obtain $O^{(n_r-1)}$.

We now describe a second algorithm to extract the coefficients $c_i^{(r)}$ from $O^{(r)}$ (Algorithm 4). As discussed previously, we know that all $c_i^{(r)}$ will be present in $O^{(r)}$ and $I^{(r+1)}$. Once again, no brute-force search is required to extract the coefficients.

Algorithm 4 Recovering $c_i^{(r)}$ from $O^{(r)}$

$j \leftarrow 1$
for $j < n-1$ **do**
  $c_{n+1-j}^{(r)} \leftarrow O_{n,n+1+j}^{(r)}$
  $c_{2n-j}^{(r)} \leftarrow O_{2n,n+1+j}^{(r)} + c_{n+1-j}^{(r)}$
  $j \leftarrow j+1$
end for
$c_{n+1}^{(r)} \leftarrow O_{n,1}^{(r)} + O_{n,n+1}^{(r)}$
$c_2^{(r)} \leftarrow O_{2n,1}^{(r)} + O_{2n,n+1}^{(r)} + c_{n+1}^{(r)}$
$c_1^{(r)} \leftarrow O_{n,n+1}^{(r)} + c_2^{(r)}$
$c_{2n}^{(r)} \leftarrow O_{2n,n+1}^{(r)} + O_{n,n+1}^{(r)}$
return $c_i^{(r)}$

Algorithm 4 can then be used to generate the self-equivalence $(O^{(n_r-1)}, I^{(n_r)})$ from $O^{(n_r-1)}$. The output external encoding $O^{(n_r)}$ can be obtained analogously to $I^{(1)}$.

Alternatively, the attacker could use the self-equivalence property of $O^{(n_r-1)}$ and $I^{(n_r)}$ to compute $I^{(n_r)}$. By the definition of a self-equivalence (Definition 7):

$$I^{(n_r)} = SL \circ O^{(n_r-1)} \circ SL^{-1}$$

Computing $SL \circ O^{(n_r-1)} \circ SL^{-1}$ for $2n$ independent input vectors of length $2n$ is enough to fully reconstruct the matrix representation of $I^{(n_r)}$.

Finally, the attacker can compute $O^{(r)}$ for $1 \leq r \leq m$ using Algorithm 3 on $\overline{M^{(r+1)}}$. This allows the attacker to recover the first m round keys from $\overline{v^{(r)}}$, and compute the master key k. Having recovered the master key and the external

encodings, the attacker can encrypt or decrypt any value, without using the encoded implementation $\overline{E_k}$.

This attack shows that, even with relatively limited capabilities, a white-box Speck implementation using only linear encodings is insecure against key recovery attacks. In particular, it is not necessary to inspect or modify the execution of the white-box implementation. Furthermore, recovering the encodings is possible using only the information revealed by a single encoded affine layer.

After considering this disappointing result, one might wonder if linear self-equivalences could still be securely applied to ARX ciphers with different affine layers. Perhaps, if $M^{(r)}$ would be shaped differently, this attack could be prevented. However, after randomly generating invertible matrices $M^{(r)}$, and computing the encoded version $\overline{M^{(r)}} = O^{(r)}M^{(r)}I^{(r)}$, we found that coefficients of the input self-equivalence encoding could still be extracted from $\overline{M^{(r)}}$ without additional effort. Evidently, this vulnerability is inherent to the shape of the linear self-equivalences of the modular addition.

5.4 Security analysis of affine self-equivalences

Knowing that a white-box Speck implementation using only linear encodings is insecure, we can try to extend these attacks to the variant using affine encodings. We start by updating the equations for $\overline{M^{(r)}}$ and $\overline{v^{(r)}}$ with affine self-equivalence encodings $(I^{(r)}, i^{(r)})$ and $(O^{(r)}, o^{(r)})$:

$$\begin{aligned}
\overline{M^{(0)}} &= M^{(0)} \\
\overline{v^{(0)}} &= v^{(0)} \\
\overline{M^{(r)}} &= O^{(r)}M^{(r)}I^{(r)}, \text{ for } 1 \leq r \leq n_r \\
\overline{v^{(r)}} &= O^{(r)}(v^{(r)} \oplus M^{(r)}i^{(r)}) \oplus o^{(r)}, \text{ for } 1 \leq r \leq n_r
\end{aligned}$$

Once again, there are two approaches to recover the round key from $\overline{v^{(r)}}$.

We previously showed that just a single key bit from $k^{(r)}$ is hidden when only linear encodings are used. However, with the introduction of affine self-equivalences, the definition of $\overline{v^{(r)}}$ also includes addition with constants $i^{(r)}$ and $o^{(r)}$. As a result, one might expect that recovery of $k^{(r)}$ will be more difficult.

On the other hand, there are no explicit changes to the definition of $\overline{M^{(r)}}$, though we know that the security of $\overline{M^{(r)}}$ depends on the shape and density of $I^{(r)}$ and $O^{(r)}$. The shape of these matrices will be discussed in the next section.

5.4.1 Recovering key bits from $\overline{v^{(r)}}$

As mentioned in Section 4.4, affine self-equivalences are generated by composing so-called type 1 and type 2 affine self-equivalences. This increases the size of the affine self-equivalence search space, thus improving the resistance to brute-force search. For a word size n, the methods to generate type 1 or type 2 affine self-equivalences

both require $2n + 7$ free coefficients. For the sake of simplicity, we will consider the shape of type 1 and type 2 affine self-equivalences separately.

Let $((O_1^{(r)}, o_1^{(r)}), (I_1^{(r+1)}, i_1^{(r+1)}))$ be a type 1 affine self-equivalence, generated using the secret coefficients $c_{1,i}^{(r)}$, with $1 \leq i \leq 2n + 7$. The shape of this self-equivalence, as $2n \times 2n$ matrices and vectors of length $2n$, is given by:

$$O_1^{(r)} = \left[\begin{array}{cccccc|cccccc} 1 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ c & 1 & 0 & 0 & 0 & 0 & c & 0 & \dots & 0 & 0 & 0 \\ \vdots & 0 & \ddots & 0 & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ c & 0 & 0 & 1 & 0 & 0 & c & 0 & \dots & 0 & 0 & 0 \\ c & 0 & \dots & 0 & 1 & 0 & c & 0 & \dots & 0 & 0 & 0 \\ c & 0 & \dots & 0 & c & 1 & c & c & \dots & c & c & c \\ \hline 0 & 0 & \dots & 0 & 0 & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ c & 0 & \dots & 0 & 0 & 0 & c & 1 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & 0 & \ddots & 0 & \vdots & \vdots \\ c & 0 & \dots & 0 & 0 & 0 & c & 0 & 0 & 1 & 0 & 0 \\ c & 0 & \dots & 0 & 0 & 0 & c & 0 & \dots & 0 & 1 & 0 \\ c & 0 & \dots & 0 & 0 & 0 & c & c & \dots & c & c & 1 \end{array}\right]$$

$$o_1^{(r)} = \left[\begin{array}{cccccc|cccccc} c & c & \dots & c & c & c & c & c & \dots & c & c & c \end{array}\right]$$

$$I_1^{(r+1)} = \left[\begin{array}{cccccc|cccccc} 1 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ c & 1 & 0 & 0 & 0 & 0 & c & 0 & \dots & 0 & 0 & 0 \\ \vdots & 0 & \ddots & 0 & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ c & 0 & 0 & 1 & 0 & 0 & c & 0 & \dots & 0 & 0 & 0 \\ c & 0 & \dots & 0 & 1 & 0 & c & 0 & \dots & 0 & 0 & 0 \\ c & 0 & \dots & 0 & c & c & c & c & \dots & c & c & c \\ \hline 0 & 0 & \dots & 0 & 0 & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ c & 0 & \dots & 0 & 0 & 0 & c & 1 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & 0 & \ddots & 0 & \vdots & \vdots \\ c & 0 & \dots & 0 & 0 & 0 & c & 0 & 0 & 1 & 0 & 0 \\ c & 0 & \dots & 0 & 0 & 0 & c & 0 & \dots & 0 & 1 & 0 \\ c & 0 & \dots & 0 & c & 0 & c & c & \dots & c & c & c \end{array}\right]$$

$$i_1^{(r+1)} = \left[\begin{array}{cccccc|cccccc} c & c & \dots & c & c & c & c & c & \dots & c & c & c \end{array}\right]$$

Here, c denotes a symbolic value which depends on one or more coefficients $c_{1,i}^{(r)}$. $O_1^{(r)}$ and $I_1^{(r+1)}$ each contain $2n + 4$ coefficients, while $o_1^{(r)}$ contains only 8 coefficients and $i_1^{(r+1)}$ contains all $2n + 7$. Furthermore, the coefficients $c_{1,1}^{(r)}$ and $c_{1,2}^{(r)}$ only occur in the constant vectors. As an immediate result, not all $2n + 7$ coefficients $c_{1,i}^{(r)}$ will be present in an encoded matrix $\overline{M^{(r)}}$.

Now, let $((O_2^{(r)}, o_2^{(r)}), (I_2^{(r+1)}, i_2^{(r+1)}))$ be a type 2 affine self-equivalence, generated using the secret coefficients $c_{2,i}^{(r)}$, with $1 \leq i \leq 2n+7$. The shape of this self-equivalence, as $2n \times 2n$ matrices and vectors of length $2n$, is given by:

$$O_2^{(r)} = \left[\begin{array}{cccccc|cccccc} c & 0 & \dots & 0 & 0 & 0 & c & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & 0 & \ddots & 0 & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ c & 0 & \dots & 0 & c & 1 & c & c & \dots & c & c & c \\ \hline c & 0 & \dots & 0 & 0 & 0 & c & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & 0 & \ddots & 0 & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 \\ c & 0 & \dots & 0 & 0 & 0 & c & c & \dots & c & c & 1 \end{array}\right]$$

$$o_2^{(r)} = \left[\begin{array}{cccccc|cccccc} c & 0 & \dots & 0 & c & c & c & 0 & \dots & 0 & c & c \end{array}\right]$$

$$I_2^{(r+1)} = \left[\begin{array}{cccccc|cccccc} c & 0 & \dots & 0 & 0 & 0 & c & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & 0 & \ddots & 0 & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ c & 0 & \dots & 0 & c & c & c & c & \dots & c & c & c \\ \hline c & 0 & \dots & 0 & 0 & 0 & c & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & 0 & \ddots & 0 & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 \\ c & 0 & \dots & 0 & 0 & 0 & c & c & \dots & c & c & c \end{array}\right]$$

$$i_2^{(r+1)} = \left[\begin{array}{cccccc|cccccc} c & 0 & \dots & 0 & c & c & c & 0 & \dots & 0 & c & c \end{array}\right]$$

Once again, c denotes a symbolic value which depends on one or more coefficients $c_{2,i}^{(r)}$. $O_1^{(r)}$ and $I_1^{(r+1)}$ contain $2n+4$ and $2n+3$ coefficients respectively, while $o_1^{(r)}$ and $i_1^{(r+1)}$ contain only 8 and 10 coefficients. This also means that not all $2n+7$ coefficients $c_{2,i}^{(r)}$ will be present in an encoded matrix $\overline{M^{(r)}}$.

Individually, the matrices for type 1 and type 2 affine self-equivalences are still very sparse. On top of this, the constant vectors for type 2 affine self-equivalences also contain many zero entries. Although the self-equivalences are composed before $v^{(r)}$ is encoded, the security of $k^{(r)}$ might still be compromised.

By computing the symbolic equations of each element in $\overline{v^{(r)}}$ and performing Gaussian elimination on this set, we found that candidate round keys could be computed by guessing the values of only two bits. An attacker could extend this to candidate master keys by guessing 2^{2m} values for m consecutive round keys. While this is a modest increase in security compared to the linear encodings, this is still substantially less than the 2^{mn} guesses required without additional information. For example, computing all candidate master keys for Speck128/256 requires guessing at most 2^8 values, down from 2^{256}.

Clearly, using (composed) affine self-equivalences to encoded constant vectors suffers from the same weakness as using linear self-equivalences. Even though the external encodings are not recoverable using this method, candidate master keys can be computed with limited effort. For this reason, we conclude that our white-box Speck method is insecure if external encodings are not used.

5.4.2 Recovering encodings

For our final approach, we aim to recover the secret coefficients, used to generate $I^{(r)}$ and $O^{(r)}$, from the encoded matrix $\overline{M^{(r)}}$. Because affine self-equivalences are constructed by composing type 1 and type 2 affine self-equivalences, $2 \times (2n + 7)$ coefficients are necessary to regenerate these encodings. However, as mentioned previously, not all $2n + 7$ coefficients used to generate a type 1 or type 2 affine self-equivalence are contained in the input and output encodings.

To simplify our analysis, without increasing the capabilities of an attacker, we consider the scenario where $\overline{M^{(r-1)}}$, $\overline{M^{(r)}}$, and $\overline{M^{(r+1)}}$ are known. The goal is to recover either $c_{1,i}^{(r-1)}$ and $c_{2,i}^{(r-1)}$, or $c_{1,i}^{(r)}$ and $c_{2,i}^{(r)}$, for all $1 \leq i \leq 2n + 7$. This would allow an attacker to regenerate the self-equivalence encodings for $(I^{(r)}, i^{(r)})$ or $(O^{(r)}, o^{(r)})$, recover the round key, and compute the external encodings.

We computed the symbolic equations of every element in $\overline{M^{(r-1)}}$, $\overline{M^{(r)}}$, and $\overline{M^{(r+1)}}$, and performed Gaussian elimination on this set of equations. We found that $(I^{(r)}, i^{(r)})$ or $(O^{(r)}, o^{(r)})$ could be recovered by guessing the values of $2n + 12$ bits. In other words, at most 2^{2n+12} guesses would be required to break the security of the implementation, using this approach. For Speck word sizes of $n = 32$, $n = 48$, and $n = 64$, this quickly becomes infeasible.

It seems that composing type 1 and type 2 affine self-equivalences leads to a resistance against this algebraic approach. Although the brute-force search space is halved, the search space was already doubled by introducing $2n + 7$ coefficients for both types. Consequently, a white-box Speck implementation using affine encodings and external encodings might be secure against key recovery attacks. Still, it is not impossible to imagine a more sophisticated approach, which reduces the brute-force search space to a more manageable level.

Chapter 6

Implementation

In previous chapters, we discussed the theoretical foundations of our method to construct white-box Speck implementations. To research the practical viability of this method, we also implemented a program to generate white-box Speck code. This implementation is publicly available in our GitHub repository[1]. In Section 6.2, we will introduce the architecture of this project and explain various design decisions. Then, in Section 6.3, we will examine some alternative strategies to generate the white-box Speck code. Finally, we end with a performance-based comparison of these different strategies in Section 6.4.

6.1 Introduction

Creating a functional implementation of a proposed method to construct white-box cryptographic implementations has multiple important advantages. For example, it encourages further cryptanalysis of the method. Theoretical white-box attacks on the method can easily be implemented and tested in a real-world setting. Furthermore, even if the method is eventually broken using algebraic or side-channel attacks, the implementation can still be useful to test more efficient key recovery attacks and investigate their applicability to other white-box implementations.

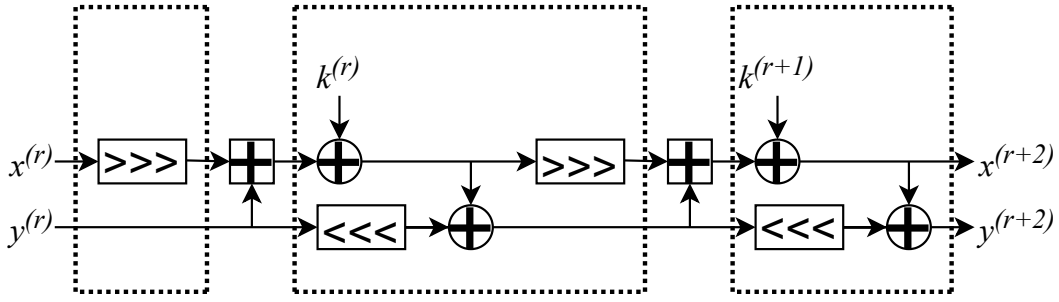

Figure 6.1: Diagram of two Speck encryption rounds

[1] https://github.com/jvdsn/white-box-speck

A functional protected implementation also provides a way to compare the protected implementation with its unprotected counterpart and benchmark the performance impact of the white-box protection. This property in particular is the main motivation behind the implementation of our method.

A diagram of the Speck encryption function can be found on Figure 6.1. The dotted boxes indicate the affine layers of the Speck rounds, consisting of bitwise XOR operations and bitwise circular shifts. These operations can be performed very quickly, commonly consuming 1 CPU cycle or less [46]. However, as mentioned in Section 4.2, a protected version of Speck is constructed by viewing the cipher as a substitution-permutation network (SPN) and encoding the affine layers using self-equivalences. Because of this, x and y are replaced by a vector of little-endian bits (xy) and the affine layers are implemented using a matrix-vector product and vector addition (Figure 6.2).

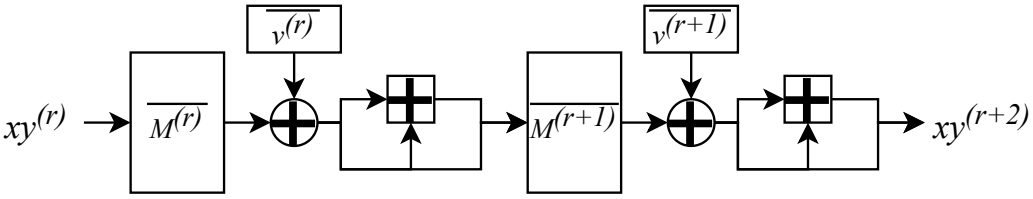

Figure 6.2: Diagram of two white-box Speck encryption rounds

It is reasonable to assume these differences will cause a significant overhead, both in terms of execution time and disk space usage, compared to an unprotected Speck implementation. By creating a functional protected implementation, this overhead can be quantified. Furthermore, different strategies for the storage of matrices and vectors, as well as the execution of the matrix-vector product and vector addition, can mitigate this overhead. The impact of these mitigations can then be compared to the standard white-box implementation.

As mentioned in Chapter 4, this technique could in fact be applied to protect any add-rotate-xor (ARX) cipher. The affine part of a round can simply be replaced by a single matrix and a single vector, while the modular addition is composed with the appropriate self-equivalences. As a result, the strategies to save disk space and improve execution time described in this chapter could be used to optimize white-box implementations of other ARX ciphers.

6.2 Architecture

Our program to generate white-box Speck implementations[2] is written in Python, a free and open source programming language [47]. We chose Python because its

[2]Our implementation currently only supports the generation of white-box Speck encryption code. However, the existing project could easily be modified to also protect and generate a protected Speck decryption implementation. When discussing protected code in this chapter, we always refer to Speck encryption.

source code is completely portable across platforms, programming in Python is comparatively simple, and it is possible to interact with SageMath using a language interface [48]. SageMath is a free and open source mathematics package, which is used extensively for mathematical computations throughout the project.

The entire process to generate a protected implementation, excluding external encodings, consists of two major steps across three different components (Figure 6.3).

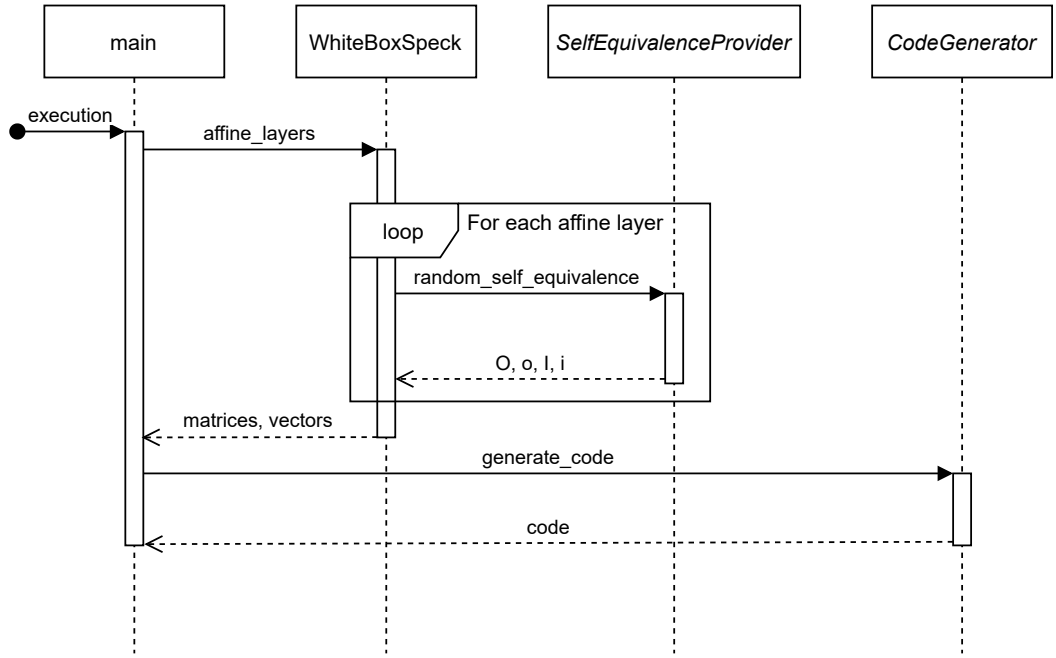

FIGURE 6.3: Sequence diagram of the white-box SPECK implementation generation

Firstly, the encoded affine layers, consisting of matrices $\overline{M^{(r)}}$ and vectors $\overline{v^{(r)}}$, are computed by the `WhiteBoxSpeck` class. This step, discussed in Section 6.2.1, supports all SPECK variants. A `SelfEquivalenceProvider` implementation can be provided to generate specific types of self-equivalences when required to protect the affine layers. The different `SelfEquivalenceProvider` implementations are described in Section 6.2.2.

After $\overline{M^{(r)}}$ and $\overline{v^{(r)}}$ are generated, they can be used to create a protected SPECK implementation. This is done by an implementation of the `CodeGenerator` class, which is described in detail in Section 6.2.3. Most implementations can generate code for any SPECK variant, however SIMD code (Section 6.3.5) only supports block sizes 32, 64, and 128.

6.2.1 Computing affine layers

The `WhiteBoxSpeck` class performs basic operations related to the SPECK cipher: determining α, β, and the amount of rounds n_r required for the word size n and the key size, perform key expansion to obtain the round keys, and generate the

matrices and vectors representing the Speck affine layers. These matrices and vectors manipulate the state vector xy using the matrix-vector product and (element-wise) vector addition, performed in $\mathbb{F}_2$. We briefly repeat the definitions for the Speck affine layers as $2n$-bit permutations on xy (Definition 10):

$$\begin{aligned} AL^{(0)}(xy) &= R_\alpha(xy) \\ AL^{(r)}(xy) &= R_\alpha(X(L_\beta(xy \oplus k'^{(r)}), \text{ for } 1 \leq r \leq n_r - 1 \\ AL^{(n_r)}(xy) &= X(L_\beta(xy \oplus k'^{(n_r)})) \end{aligned}$$

Here R_α represents a right bitwise rotation of x by α positions, L_β represents a left bitwise rotation of y by β positions, and X represents the bitwise XOR operation such that $y = x \oplus y$.

Because these functions are linear, the corresponding $2n \times 2n$ matrices can be constructed easily. Note that these matrices all operate on the little-endian bits of x and y in the state vector xy.

$$R_\alpha = \left[\begin{array}{cccccc|cccccc} 0 & \dots & 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & 0 & \ddots & 0 & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & 0 & 1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 1 & 0 & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & \ddots & 0 & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \hline 0 & \dots & 0 & 0 & \dots & 0 & 1 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & 0 & \ddots & 0 & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & \dots & 0 & 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & 0 & \ddots & 0 \\ 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & 0 & 1 \end{array}\right] \begin{array}{l} \left.\vphantom{\begin{array}{c}a\\a\\a\end{array}}\right\} n-\alpha \\ \left.\vphantom{\begin{array}{c}a\\a\\a\end{array}}\right\} \alpha \\ \left.\vphantom{\begin{array}{c}a\\a\\a\\a\\a\\a\end{array}}\right\} n \end{array}$$

$$L_\beta = \left[\begin{array}{cccccc|cccccc} 1 & 0 & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & \ddots & 0 & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & \dots & 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & 0 & \ddots & 0 & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & 0 & 1 & 0 & \dots & 0 & 0 & \dots & 0 \\ \hline 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & 0 & \ddots & 0 \\ 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & 0 & 1 \\ 0 & \dots & 0 & 0 & \dots & 0 & 1 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & 0 & \ddots & 0 & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & \dots & 0 & 0 & 0 & 1 & 0 & \dots & 0 \end{array}\right] \begin{array}{l} \left.\vphantom{\begin{array}{c}a\\a\\a\\a\\a\\a\end{array}}\right\} n \\ \left.\vphantom{\begin{array}{c}a\\a\\a\end{array}}\right\} \beta \\ \left.\vphantom{\begin{array}{c}a\\a\\a\end{array}}\right\} n-\beta \end{array}$$

$$X = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & \ddots & 0 & \vdots & \ddots & \vdots \\ 0 & 0 & 1 & 0 & \dots & 0 \\ \hline 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & \ddots & 0 & 0 & \ddots & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \begin{array}{l} \left.\vphantom{\begin{array}{c}1\\0\\0\end{array}}\right\} n \\ \left.\vphantom{\begin{array}{c}1\\0\\0\end{array}}\right\} n \end{array}$$

Finally, $k'^{(r)}$ is a vector of length $2n$ containing the key bits of the round key $k^{(r)}$ at the first n positions:

$$k'^{(r)} = \left[\begin{array}{ccc|ccc} k_1^{(r)} & \dots & k_n^{(r)} & 0 & \dots & 0 \end{array}\right]$$

For efficiency reasons, we would like to protect and store a single matrix and vector for each affine layer (Figure 6.4). We define $M^{(r)}$ and $v^{(r)}$ as follows:

$$\begin{aligned} M^{(0)} &= R_\alpha & v^{(0)} &= [0 \dots 0] \\ M^{(r)} &= R_\alpha X L_\beta, \text{ for } 1 \leq r \leq n_r - 1 & v^{(r)} &= M^{(r)} k'^{(r)}, \text{ for } 1 \leq r \leq n_r - 1 \\ M^{(n_r)} &= X L_\beta & v^{(n_r)} &= M^{(n_r)} k'^{(n_r)} \end{aligned}$$

The equations for the affine layers can then simply be written as:

$$AL^{(r)}(xy) = M^{(r)} xy \oplus v^{(r)}, \text{ for } 0 \leq r \leq n_r$$

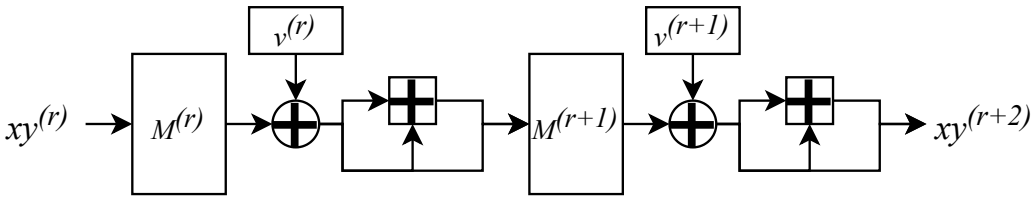

FIGURE 6.4: Diagram of two SPECK SPN encryption rounds, with affine layers replaced by a matrix-vector product and vector addition

In Section 4.2, we introduced the definition of an affine layer protected using self-equivalence encodings:

$$\overline{AL^{(r)}} = (\oplus_{o^{(r)}} \circ O^{(r)}) \circ AL^{(r)} \circ (\oplus_{i^{(r)}} \circ I^{(r)})$$

We can apply this definition to obtain the protected matrices $\overline{M^{(r)}}$ and vectors $\overline{v^{(r)}}$. However, because $AL^{(0)}$ does not contain any key material ($v^{(0)} = [0 \dots 0]$) we can skip this layer. Instead, we start the protection at $AL^{(1)}$:

$$\begin{aligned} \overline{M^{(0)}} &= M^{(0)} \\ \overline{v^{(0)}} &= v^{(0)} \\ \overline{M^{(r)}} &= O^{(r)} M^{(r)} I^{(r)}, \text{ for } 1 \leq r \leq n_r \\ \overline{v^{(r)}} &= O^{(r)}(v^{(r)} \oplus M^{(r)} i^{(r)}) \oplus o^{(r)}, \text{ for } 1 \leq r \leq n_r \end{aligned}$$

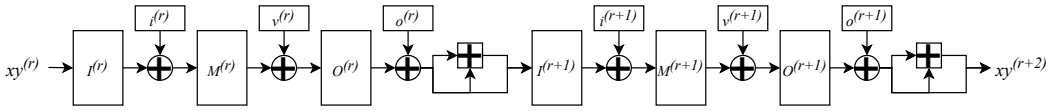

Figure 6.5: Diagram of two Speck SPN encryption rounds, with self-equivalences applied to the rounds

Note that $(I^{(1)}, i^{(1)})$ and $(O^{(n_r)}, o^{(n_r)})$ are the external encodings. These encodings are simply generated by sampling random invertible matrices and vectors, and output alongside the protected Speck implementation.

We now have all information required to compute the protected matrices and vectors representing the Speck affine layers. In our implementation, the `affine_layers` function in `WhiteBoxSpeck` takes the external encodings and an instance of the `SelfEquivalenceProvider`, computes the matrices $M^{(r)}$ and vectors $v^{(r)}$, and then applies a random self-equivalence from the `SelfEquivalenceProvider` for each matrix and vector when necessary. In the next subsection, we explain how these random self-equivalence encodings are generated.

6.2.2 Generating self-equivalences

In our project, the `SelfEquivalenceProvider` class has four subclasses which contain the `random_self_equivalence` function to generate self-equivalences (Figure 6.6). This function returns a tuple (A, a, B, b) such that $((A, a), (B, b))$ is a self-equivalence for the modular addition (see Section 4.1).

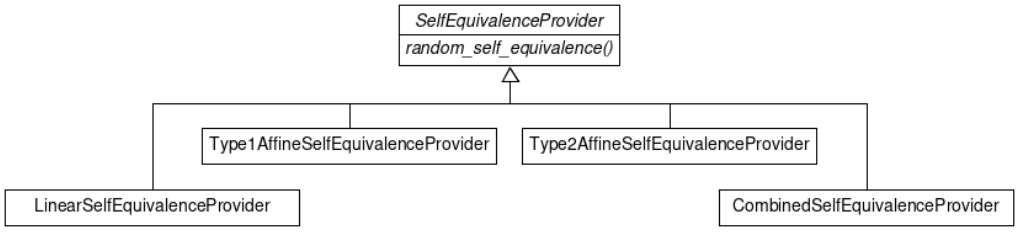

Figure 6.6: UML diagram of the classes responsible for generating self-equivalences

`LinearSelfEquivalenceProvider` generates linear self-equivalences using the method described in Section 4.3. As a result, the a and b vectors returned by `random_self_equivalence` will always be zero vectors.

`Type1AffineSelfEquivalenceProvider` generates type 1 affine self-equivalences, as described in Section 4.4.

`Type2AffineSelfEquivalenceProvider` generates type 2 affine self-equivalences, as described in Section 4.4.

`CombinedSelfEquivalenceProvider` contains a collection of delegate self-equivalence providers. When the `random_self_equivalence` function is called, the self-equivalences generated by each of the delegates are composed to increase the self-equivalence search space.

6.2.3 Code generation

Code generation is handled by the `generate_code` function in implementations of the abstract `CodeGenerator` class. In total, we implemented 6 `CodeGenerator` subclasses, one of which corresponds to a default implementation, and 5 others providing different code generation strategies (detailed in Section 6.3). Figure 6.7 contains an overview of the `CodeGenerator` subclasses. In this section we will focus on the code generated by the `DefaultCodeGenerator`.

Our `CodeGenerator` implementations exclusively generate C source code. We chose the C programming language because it is widely used, provides fast low-level memory control, and contains a convenient interface for single instruction, multiple data (SIMD) functions (Section 6.3.5). Of course, any subclass of `CodeGenerator` can freely choose to return source or compiled code for other programming languages.

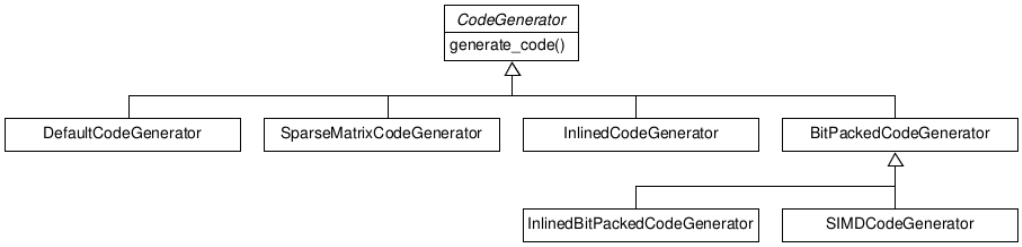

FIGURE 6.7: UML diagram of the classes responsible for code generation

The generated C code to perform white-box SPECK encryption follows the same intuitive pattern as simple SPN cipher implementations. For each round, a modular addition and affine transformation is performed, except for the final round, which consists only of the affine transformation. Of course, the white-box SPECK encryption algorithm operates on vectors of bits instead of integers, so the input x and y should be converted to bits first. Similarly, xy should be converted back to integers after encryption. No key expansion is necessary, as the round keys $k^{(r)}$ are implicitly stored in the vectors $v^{(r)}$.

Algorithm 5 relies on five subroutines: functions to convert to and from bits, a function to perform the modular addition on xy, a function to perform the matrix-vector product, and a function to perform the vector addition. As we consider the conversion to and from binary to be trivial, we will not elaborate on the implementation of these functions.

The other three functions use a standard textbook implementation. For example, the modular addition simply performs the addition with carry algorithm on each

Algorithm 5 White-box Speck encryption

$xy \leftarrow$ input plaintext x, y converted to bits
$r \leftarrow 0$
for $r < n_r$ **do**
  Multiply xy by the round matrix $\overline{M^{(r)}}$
  Add the round vector $\overline{v^{(r)}}$ to xy
  Perform the modular addition on xy
  $r \leftarrow r + 1$
end for
Multiply xy by the final round matrix $\overline{M^{(n_r)}}$
Add the final round vector $\overline{v^{(n_r)}}$ to xy
return bits xy converted to output ciphertext

individual bit, ignoring the final carry to perform the modular reduction. The generated C code for this function can be found in Listing 6.1.

```
void modular_addition(uint8_t xy[BLOCK_SIZE]) {
    uint8_t carry = 0;
    for (size_t i = 0; i < WORD_SIZE; i++) {
        xy[i] = xy[i] + xy[WORD_SIZE + i] + carry;
        carry = xy[i] > 1;
        xy[i] &= 1;
    }
}
```

Listing 6.1: Generated C code to compute the modular addition

In the case of the matrix-vector product, two `for` loops are used to compute the resulting vector (Listing 6.2).

```
void matrix_vector_product(uint8_t matrix[BLOCK_SIZE][BLOCK_SIZE],
    ↪ uint8_t xy[BLOCK_SIZE], uint8_t res[BLOCK_SIZE]) {
    for (size_t i = 0; i < BLOCK_SIZE; i++) {
        for (size_t j = 0; j < BLOCK_SIZE; j++) {
            res[i] ^= matrix[i][j] * xy[j];
        }
    }
}
```

Listing 6.2: Generated C code to compute the matrix-vector product

For the vector addition, the generated code performs an XOR operation for each bit in the vector (Listing 6.3).

Finally, apart from the definitions and implementations of these subroutines, the required data (matrices $\overline{M^{(r)}}$ and vectors $\overline{v^{(r)}}$) will also have to be stored in the C source code. A straightforward way of storing a matrix in C is to use a

```
void vector_addition(uint8_t vector[BLOCK_SIZE], uint8_t xy[
    ↪ BLOCK_SIZE]) {
    for (size_t i = 0; i < BLOCK_SIZE; i++) {
        xy[i] ^= vector[i];
    }
}
```

Listing 6.3: Generated C code to compute the vector addition

two-dimensional array: storing each row as an array of the elements in an enclosing array to represent the full matrix. A vector can be stored by simply using a single one-dimensional array. In total, $n_r + 1$ matrices and $n_r + 1$ are generated by the `DefaultCodeGenerator`.

6.3 Code generation strategies

Although the method described in the previous section generates correct and functional C code, this code is far from optimal. In this section we will introduce and compare different techniques to improve the efficiency of the generated C code in four different metrics: the disk space used to store the matrices $\overline{M^{(r)}}$, the disk space used to store the vectors $\overline{v^{(r)}}$, the execution time of the matrix-vector product, and the execution time of the vector addition. When quantifying the required disk space or execution time in this section, we will always denote the Speck word size as n, unless otherwise indicated.

The smallest supported integer type in the C programming language has a size of 8 bits (1 byte) [49, p. 20]. Consequently, storing a matrix $\overline{M^{(r)}}$ as a two-dimensional array requires at least $(2n)^2$ bytes of disk space. Similarly, storing a vector $\overline{v^{(r)}}$ as a one-dimensional array will take $2n$ bytes of disk space.

Furthermore, the standard implementation of the matrix-vector product consists of two `for` loops of $2n$ entries, bringing the total number of iterations to $(2n)^2$. The vector addition requires only one `for` loop of $2n$ entries, resulting in only $2n$ iterations.

6.3.1 Sparse matrix code generation

Because the entries of $\overline{M^{(r)}}$ are in $\mathbb{F}_2$, one could consider storing only the nonzero entries to save disk space. The other entries are then implicitly known to be 0. We formalize this by introducing the sparse matrix representation.

Definition 14 (Sparse matrix representation)**.** *The sparse matrix representation of the $2n \times 2n$ matrix $\overline{M^{(r)}}$ is a list of coordinate pairs (i, j), with $1 \leq i, j \leq 2n$, such that $\overline{M^{(r)}}_{i,j} = 1$.*

Storing $\overline{M^{(r)}}$ as a sparse matrix requires $2m$ bytes of disk space, where m is the number of nonzero entries, assuming coordinate pair (i, j) can be stored in 2 bytes.

Because the largest Speck word size is 64, the matrix will contain at most 128×128 entries, so $0 \leq i, j < 128$, within the limit of 2 bytes. As mentioned before, storing this matrix as a two-dimensional array requires $(2n)^2$ bytes of disk space. From this, we can derive that using the sparse matrix representation will be the better choice if $m < \frac{(2n)^2}{2}$.

Because not all affine layers contain key material, not all matrices $M^{(r)}$ are encoded using self-equivalences:

$$\overline{M^{(0)}} = M^{(0)}$$
$$\overline{M^{(r)}} = O^{(r)} M^{(r)} I^{(r)}, \text{ for } 1 \leq r \leq n_r$$

However, a large majority of the matrices will follow the form for $\overline{M^{(r)}} = O^{(r)} M^{(r)} I^{(r)}$, so computing m for these matrices will provide the most information. We generated 10,000 matrices of this form for each Speck word size, to determine the average number of nonzero entries. Figure 6.8 plots the results, as well as a linear regression on this data, which indicates that m is linearly related to the word size n.

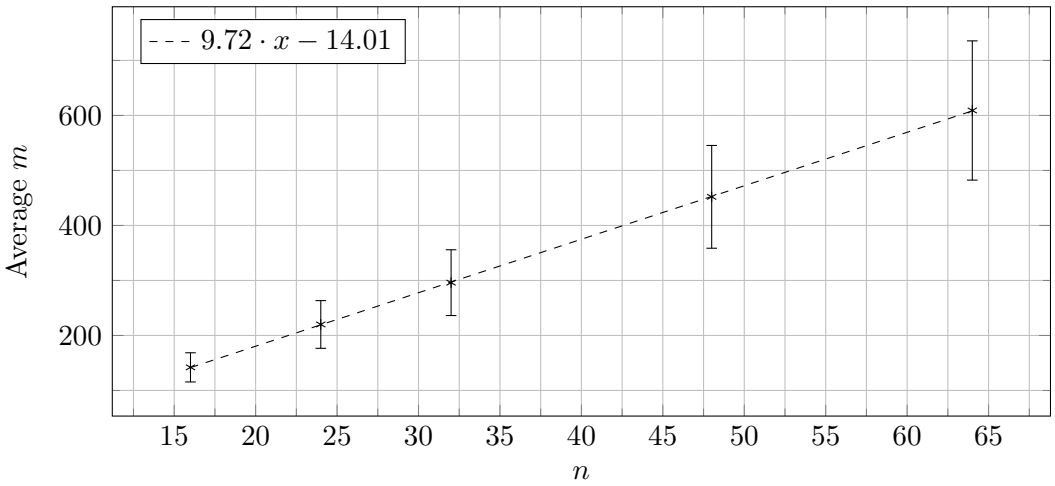

Figure 6.8: Average number of nonzero entries in $\overline{M^{(r)}}$

	$n = 16$	$n = 24$	$n = 32$	$n = 48$	$n = 64$
Average m	142	220	296	452	609
Disk space usage (bytes)	284	440	592	904	1218
Disk space saved	72.27%	80.90%	85.55%	90.19%	92.57%

Table 6.1: Average number of nonzero entries, disk space usage, and disk space saved using the sparse matrix representation

From m we can now compute the disk space usage and saved disk space using the sparse matrix representation (Table 6.1). As shown by the experimental results, using the sparse matrix representation to store $\overline{M^{(r)}}$ is beneficial for all word sizes, on

average. Furthermore, because of the linear relation between m and n, the percentage of saved disk space increases quadratically as the word size increases, saving up to 92.57% in disk space on average.

In addition to reducing the disk space used by the generated C code, using the sparse matrix representation also simplifies the matrix-vector product. The sparse matrix representation only contains the nonzero entries of $\overline{M^{(r)}}$, exactly the entries needed to compute the matrix-vector product. As a result, instead of $(2n)^2$ iterations, only m iterations of a `for` loop are necessary. Listing 6.4 contains the generated C code.

```
void matrix_vector_product(uint8_t sparse_matrix[][2], uint16_t
    ↪ sparse_matrix_entries, uint8_t xy[BLOCK_SIZE], uint8_t res[
    ↪ BLOCK_SIZE]) {
    for (uint16_t i = 0; i < sparse_matrix_entries; i++) {
        res[sparse_matrix[i][0]] ^= xy[sparse_matrix[i][1]];
    }
}
```

Listing 6.4: Generated C code to compute the matrix-vector product using the sparse matrix representation

Similar to the sparse matrix representation, we can also introduce the sparse vector representation for the vectors $\overline{v^{(r)}}$.

Definition 15 (Sparse vector representation)**.** *The sparse vector representation of the vector* $\overline{v^{(r)}}$ *of length* $2n$ *is a list of coordinates* i*, with* $1 \leq i \leq 2n$*, such that* $\overline{v^{(r)}}_i = 1$.

In this case, storing $\overline{v^{(r)}}$ using the sparse vector representation requires m bytes of disk space, with m the number of nonzero entries in the vector. As before, the vector will contain at most 128 entries, so coordinates will always fit in a byte. Because storing the vector as a one-dimensional array requires $2n$ bytes of disk space, and by definition $m \leq 2n$, the sparse vector representation will always be more efficient than storing the entire vector.

To determine the efficiency of the sparse vector representation compared to storing a one-dimensional array, we generated 10,000 vectors for each Speck word size (Figure 6.9). As with the sparse matrix representation, the relationship between m and n seems to be linear.

	$n = 16$	$n = 24$	$n = 32$	$n = 48$	$n = 64$
Average m	14	21	26	39	48
Disk space usage (bytes)	14	21	26	39	48
Disk space saved	56.25%	56.25%	59.38%	59.38%	62.50%

Table 6.2: Average number of nonzero entries, disk space usage, and disk space saved using the sparse vector representation

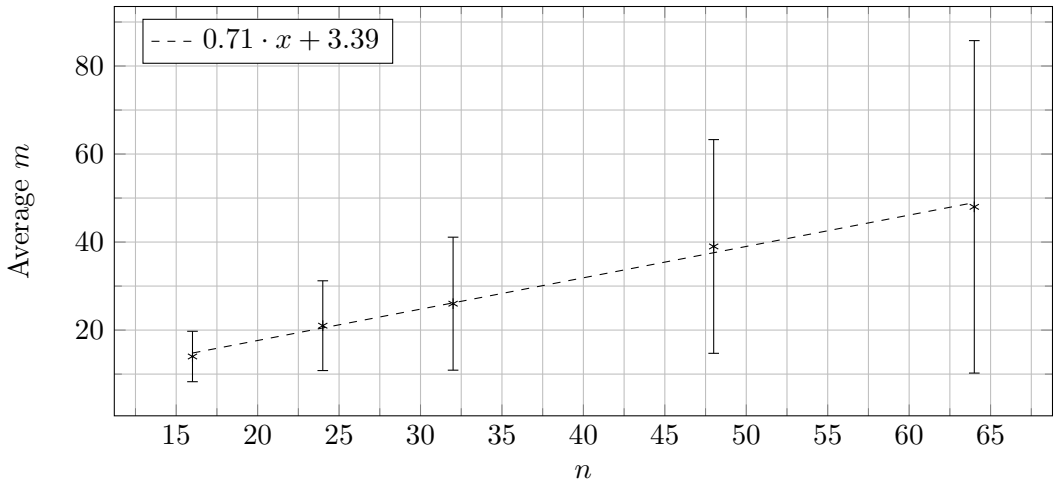

Figure 6.9: Average number of nonzero entries in $\overline{v^{(r)}}$

However, because the disk space used by a one-dimensional array is also linearly related to n, the percentage of saved disk space increases only slightly as the word size grows (Table 6.2). For $n = 64$, on average only 62.50% in disk space is saved, indicating a smaller positive impact compared to the sparse matrix representation.

The vector addition can also be modified to take advantage of the sparse vector representation. Instead of looping over every entry in the vector, only the nonzero entries have to be considered. This reduces the amount of iterations from $2n$ to m. The generated C code is shown in Listing 6.5.

```
void vector_addition(uint8_t sparse_vector[], uint8_t
    ↪ sparse_vector_entries, uint8_t xy[BLOCK_SIZE]) {
    for (uint8_t i = 0; i < sparse_vector_entries; i++) {
        xy[sparse_vector[i]] ^= 1;
    }
}
```

Listing 6.5: Generated C code to compute the vector addition using the sparse vector representation

6.3.2 Inlined code generation

In previous examples of the generated C code, the affine operations are always separated in two distinct parts. On one hand is the immutable stored data, being the matrices $\overline{M^{(r)}}$ and the vectors $\overline{v^{(r)}}$. On the other hand, we have the generic functions for the matrix-vector product or the vector addition.

However, because the contents of $\overline{M^{(r)}}$ and $\overline{v^{(r)}}$ are known before the C code is generated, it is possible to combine these two parts by generating $n_r + 1$ different functions for the matrix-vector product and for the vector addition. In the case of

the matrix-vector products, these functions will only contain the array operations for the nonzero entries in the matrix. Similarly, the functions for the vector additions only modify the positions for the nonzero entries in the vector. In this way, the data is inlined in the function implementations.

A short, incomplete example of an inlined matrix-vector product for round 2 can be found in Listing 6.6. Note that the function only accepts two parameters, the input array and the output array, compared to the three parameters in the default implementation (Listing 6.2).

```
void matrix_vector_product_2(uint8_t xy[BLOCK_SIZE], uint8_t res[
    ↪ BLOCK_SIZE]) {
    res[0] ^= xy[0] ^ xy[8] ^ xy[64];
    res[1] ^= xy[0] ^ xy[9] ^ xy[125];
    res[2] ^= xy[0] ^ xy[10] ^ xy[125];
    ...
}
```

LISTING 6.6: Generated C code to compute the matrix vector product using inlining

In general, one `xor` instruction is needed for each nonzero entry of $\overline{M^{(r)}}$. Additionally, one `movzx` instruction is required for each modification of the `xy` array. A `movzx` and a `xor` instruction take up 4 and 3 bytes of disk space, respectively. Assuming each of the $2n$ entries in `xy` is modified, the implicit storage of $\overline{M^{(r)}}$ in the function implementation would require $4(2n) + 3m$ bytes, with m the number of nonzero entries in the matrix.

Using the values for m from Table 6.1, we can try to calculate the disk space usage and saved disk space for this code generation strategy.

	$n = 16$	$n = 24$	$n = 32$	$n = 48$	$n = 64$
Average m	142	220	296	452	609
Disk space usage (bytes)	554	852	1144	1740	2339
Disk space saved	45.90%	63.02%	72.07%	81.12%	85.72%

TABLE 6.3: Average number of nonzero entries, disk space usage and disk space saved using the inlined matrix-vector product

The results in Table 6.3 indicate that inlining $\overline{M^{(r)}}$ in a function implementation for the matrix-vector product saves disk space for all word sizes, on average. The percentage of saved disk space increases as the word size increases, with up to 85.72% saved on average for $n = 64$.

The inlined implementation of a vector addition function is even simpler compared to the matrix-vector product. As shown in Listing 6.7, a simple bitwise XOR operation is performed for each nonzero entry in $\overline{v^{(r)}}$.

Once again, a single `xor` instruction is needed to perform the bitwise XOR operations, which takes up 3 bytes of disk space. Storing $\overline{v^{(r)}}$ in the inlined function requires $3m$ bytes, with m the number of nonzero entries in the vector. However,

```
void vector_addition_2(uint8_t xy[BLOCK_SIZE]) {
    xy[0] ^= 1;
    xy[2] ^= 1;
    xy[3] ^= 1;
    ...
}
```

Listing 6.7: Generated C code to compute the vector addition using inlining

the disk space required to store the vector in a one-dimensional array is $2n$ bytes, so the inlined function might be less efficient in terms of storage compared to the default implementation.

Indeed, the results in Table 6.4 (with m taken from Table 6.2) show that inlining the vectors in the vector addition functions does not save disk space for any of the Speck word sizes.

	$n = 16$	$n = 24$	$n = 32$	$n = 48$	$n = 64$
Average m	14	21	26	39	48
Disk space usage (bytes)	45	63	78	117	141
Disk space saved	-31.25%	-31.25%	-21.88%	-21.88%	-12.50%

Table 6.4: Average number of nonzero entries, disk space usage and disk space saved using the inlined vector addition

Although this strategy has mixed results in terms of disk space, its main advantage is the absence of loops in the matrix-vector product and vector addition functions. Because all necessary `xor` instructions are explicitly defined in these functions, no (conditional) jumps are needed. This allows the processor to avoid branch mispredictions and exploit instruction-level parallelism, potentially improving the execution speed of these functions. A detailed comparison of the performance impact can be found in Section 6.4.

6.3.3 Bit-packed code generation

The C standard library contains data types to store 16-bit, 32-bit, and 64-bit unsigned integers. Instead of storing the bits individually in an integer data type, we can use these larger data types to store multiple bits simultaneously, bit-packing n bits in an n-bit unsigned integer. This will greatly reduce the disk space usage and improve the execution time of the generated C code. When $n = 24$ or $n = 48$, the data must be stored in 32-bit or 64-bit unsigned integers, respectively.

We first illustrate the bit-packing for the state vector xy. By default, xy is a vector consisting of $2n$ bits, stored as 8-bit integers:

$$xy = \left[\begin{array}{ccc|ccc} xy_1 & \dots & xy_n & xy_{n+1} & \dots & xy_{2n} \end{array}\right]$$

As mentioned previously, $[xy_1 \dots xy_n]$ and $[xy_{n+1} \dots xy_{2n}]$ represent the bits in little-endian order of x and y, respectively. When these bits are packed in n-bit

integers **x** and **y**, the resulting array **xy** only contains 2 entries and the required disk space is reduced substantially:

$$\mathbf{xy} = \left[\begin{array}{c|c} \mathbf{x} & \mathbf{y} \end{array}\right]$$

We can extend this technique to the matrices $\overline{M^{(r)}}$. To simplify the implementation of the matrix-vector product, we decide to divide each row in $\overline{M^{(r)}}$ in two equal parts of n bits: the x part and the y part:

$$\overline{M^{(r)}} = \left[\begin{array}{ccc|ccc} \overline{M^{(r)}}_{1,1} & \dots & \overline{M^{(r)}}_{1,n} & \overline{M^{(r)}}_{1,n+1} & \dots & \overline{M^{(r)}}_{1,2n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \overline{M^{(r)}}_{2n,1} & \dots & \overline{M^{(r)}}_{2n,n} & \overline{M^{(r)}}_{2n,n+1} & \dots & \overline{M^{(r)}}_{2,2n} \end{array}\right]$$

Then, similar to the bit-packing of xy, each row can be transformed to an array containing 2 n-bit integers, resulting in a bit-packed matrix $\overline{\mathbf{M}^{(\mathbf{r})}}$:

$$\overline{\mathbf{M}^{(\mathbf{r})}} = \left[\begin{array}{c|c} \overline{\mathbf{M}^{(\mathbf{r})}}_{1,x} & \overline{\mathbf{M}^{(\mathbf{r})}}_{1,y} \\ \vdots & \vdots \\ \overline{\mathbf{M}^{(\mathbf{r})}}_{2n,x} & \overline{\mathbf{M}^{(\mathbf{r})}}_{2n,y} \end{array}\right]$$

Finally, the vectors $\overline{v^{(r)}}$ are also divided in x and y parts consisting of n bits:

$$\overline{v^{(r)}} = \left[\begin{array}{ccc|ccc} \overline{v^{(r)}}_1 & \dots & \overline{v^{(r)}}_n & \overline{v^{(r)}}_{n+1} & \dots & \overline{v^{(r)}}_{2n} \end{array}\right]$$

Analogous to the bit-packing of xy, $\overline{v^{(r)}}$ becomes a bit-packed array $\overline{\mathbf{v}^{(\mathbf{r})}}$, containing 2 n-bit integers representing the x and y parts of $\overline{v^{(r)}}$.

$$\overline{\mathbf{v}^{(\mathbf{r})}} = \left[\begin{array}{c|c} \overline{\mathbf{v}^{(\mathbf{r})}}_x & \overline{\mathbf{v}^{(\mathbf{r})}}_y \end{array}\right]$$

For $n = 16$, $n = 32$, and $n = 64$, the bit-packing method reduces the required disk space by a factor of 8, or 87.50%, a substantial change only matched by the sparse matrix representation from Section 6.3.1. Moreover, this reduction is present in the storage of both $\overline{M^{(r)}}$ and $\overline{v^{(r)}}$. When $n = 24$ or $n = 48$, the disk space is reduced by a factor of 6, or 83.33%.

Bit-packing xy, $\overline{M^{(r)}}$, and $\overline{v^{(r)}}$ requires us to modify the code to compute the matrix-vector product and vector addition. It is possible to simply generate code to convert the bit-packed integers to bits, perform the operations on these bits using the default implementation, and convert the bits back to bit-packed integers. Presumably, this method would be very inefficient. Instead, we can work directly with the bit-packed integers.

We first have to introduce the parity function, which will be used to simplify the result.

Definition 16 (Parity function)**.** *The parity function p computes the parity of an integer. The parity is equal to 0 if the binary representation of the integer contains an even number of ones, otherwise the parity is equal to 1:*

$$p(x) = \bigoplus_{i=1}^{n} b_i \text{ with } b_i \text{ the } i\text{'th bit of } x$$

Starting with the standard definition of the matrix-vector product, the result at index i is calculated as follows:

$$\begin{aligned}\texttt{res}_i &= \bigoplus_{j=1}^{2n} (\overline{M^{(r)}}_{j,1} \times xy_j) \\ &= \bigoplus_{j=1}^{n} (\overline{M^{(r)}}_{j,1} \times xy_j) \oplus \bigoplus_{j=1}^{n} (\overline{M^{(r)}}_{n+j,1} \times xy_{n+j})\end{aligned}$$

Because the entries in $\overline{M^{(r)}}$ and xy are bits, the element-wise multiplication is equivalent to the bitwise AND operation on the bit-packed counterparts $\overline{\mathbf{M^{(r)}}}$ and $\mathbf{xy}$. Furthermore, the parity function p can be used in place of the bitwise XOR operations on the bits. This results in:

$$\begin{aligned}\texttt{res}_i &= p(\overline{\mathbf{M^{(r)}}}_{i,x} \;\&\; \mathbf{x}) \oplus p(\overline{\mathbf{M^{(r)}}}_{i,y} \;\&\; \mathbf{y}) \\ &= p((\overline{\mathbf{M^{(r)}}}_{i,x} \;\&\; \mathbf{x}) \oplus (\overline{\mathbf{M^{(r)}}}_{i,y} \;\&\; \mathbf{y}))\end{aligned}$$

Unfortunately, there is no operator or function in the C programming language to compute the parity of an integer. However, compilers often have built-in functions for frequently used operations such as the parity function. We use the GNU Compiler Collection (GCC), which provides 3 built-in functions [50], listed in Table 6.5. Although these functions do not provide the same performance as a dedicated instruction, they are considerably faster compared to an implementation written in C.

n	**Function**
16	`__builtin_parity`
24 and 32	`__builtin_parityl`
48 and 64	`__builtin_parityll`

Table 6.5: Built-in parity functions provided by GCC

Using the appropriate built-in function, we can translate the definition of the matrix-vector product to C code (Listing 6.8). Instead of $(2n)^2$ iterations, the bit-packed version consists of one `for` loop with n iterations (but with simultaneous modification of the x and y part of $\mathbf{xy}$).

The vector addition can be performed by computing the bitwise XOR operation on the two bit-packed integers, equivalent to computing the XOR operation on the individual bits. This results in only two bitwise XOR operations, compared to the $2n$ iterations (and XOR operations) in the default implementation. The generated C code can be found in Listing 6.9.

Finally, the usage of $\mathbf{xy}$ instead of xy also simplifies the general structure of the generated C code. Because the rounds operate on 2 n-bit integers instead of $2n$ bits,

```
void matrix_vector_product(WORD_TYPE matrix[BLOCK_SIZE][2],
    ↪ WORD_TYPE xy[2], WORD_TYPE res[2]) {
    for (size_t i = WORD_SIZE; i--> 0;) {
        res[0] = (res[0] << 1) | ((WORD_TYPE) WORD_PARITY_FUNCTION
            ↪ ((matrix[i][0] & xy[0]) ^ (matrix[i][1] & xy[1])));
        res[1] = (res[1] << 1) | ((WORD_TYPE) WORD_PARITY_FUNCTION
            ↪ ((matrix[WORD_SIZE + i][0] & xy[0]) ^ (matrix[
            ↪ WORD_SIZE + i][1] & xy[1])));
    }
}
```

LISTING 6.8: Generated C code to compute the matrix-vector product using bit-packing

```
void vector_addition(WORD_TYPE vector[2], WORD_TYPE xy[2]) {
    xy[0] ^= vector[0];
    xy[1] ^= vector[1];
}
```

LISTING 6.9: Generated C code to compute the vector addition using bit-packing

it is no longer necessary to convert to and from bits, before and after the encryption. On top of this, the modular addition (Listing 6.10) can be implemented by simply using the C addition operator and a bitmask: any overflow will automatically result in a modular reduction [49, p. 35]. Together, these changes result in compact and conceptually simple code, comparable to an unprotected SPECK implementation.

```
void modular_addition(WORD_TYPE xy[2]) {
    xy[0] = (xy[0] + xy[1]) & WORD_MASK;
}
```

LISTING 6.10: Generated C code to compute the modular addition using bit-packing

6.3.4 Inlined bit-packed code generation

In Section 6.3.2 we introduced the inlined code generation strategy, which generates different matrix-vector product and vector addition functions by implicitly storing $\overline{M^{(r)}}$ and $\overline{v^{(r)}}$ in the function implementation. One might wonder if it could be possible to combine this technique with the bit-packing method from the previous section. In the inlined bit-packed code generation strategy, the state vector xy is bit-packed as described previously, but $\overline{M^{(r)}}$ and $\overline{v^{(r)}}$ are not stored explicitly in the source code.

In the inlined implementation of the matrix-vector product (Listing 6.11), bits are extracted from the bit-packed vector, and added together to compute entry i in the result vector. This entry is then bit-packed again to obtain the bit-packed result.

```
void matrix_vector_product_2(WORD_TYPE xy[2], WORD_TYPE res[2]) {
    res[0] |= (((xy[0] >> 0) & 1) ^ ((xy[0] >> 8) & 1) ^ ((xy[1]
        ↪ >> 0) & 1)) << 0;
    res[0] |= (((xy[0] >> 0) & 1) ^ ((xy[0] >> 9) & 1) ^ ((xy[1]
        ↪ >> 61) & 1)) << 1;
    res[0] |= (((xy[0] >> 0) & 1) ^ ((xy[0] >> 10) & 1) ^ ((xy[1]
        ↪ >> 61) & 1)) << 2;
    ...
}
```

Listing 6.11: Generated C code to compute the matrix vector product using inlined bit-packing

Evidently, this inlining introduces additional instruction overhead compared to both the original inlined code generation and the bit-packed code generation.

In general, one `mov` instruction (3 bytes on average), one `shr` instruction (4 bytes), and one `xor` (3 bytes) are required for each nonzero entry in $\overline{M^{(r)}}$. Furthermore, one `shl` (4 bytes), one `and` (3 bytes on average), one `or` (3 bytes), and one `mov` instruction are also needed for each modification of the result vector. Assuming each row in the result vector is modified, the implicit storage of $\overline{M^{(r)}}$ would require $13(2n) + 10m$ bytes, where m is the number of nonzero entries in the matrix.

	$n = 16$	$n = 24$	$n = 32$	$n = 48$	$n = 64$
Average m	142	220	296	452	609
Disk space usage (bytes)	1836	2824	3792	5768	7754
Disk space saved	-79.30%	-22.57%	7.42%	37.41%	52.67%

Table 6.6: Average number of nonzero entries, disk space usage and disk space saved using the bit-packed inlined matrix-vector product

Table 6.6 shows the amount of disk space that could be saved by using this technique, compared to the default implementation. Again, we used the average number of nonzero entries from Section 6.3.1. Although the required disk space for inlined bit-packed is linear in n, the constants are too large to save any disk space compared to the quadratic growth, $(2n)^2$, of the default implementation for smaller word sizes. Only starting at $n = 32$ does this strategy have a positive impact on storage, and the best average gains are modest compared to previous strategies. For $n = 64$, the inlined code generation saves 85.72%, and the bit-packing method saves 87.50%, compared to only 52.67% saved in this case.

Inlining the vector addition results in the bit-packed vector constants being stored inside the function implementation (Listing 6.12). Consequently, this method does not change the disk space used to store the vectors $\overline{v^{(r)}}$. At best, some array load instructions can be avoided. However, depending on the word size and target platform, these constants will have to be stored in the assembly `data` segment, similar to array contents. In this case, there is no technical difference with the bit-packed

code from Listing 6.9. For this reason, we do not expect any significant disk space or execution time impact caused by inlining the vector addition.

```
void vector_addition_2(WORD_TYPE xy[2]) {
    xy[0] ^= WORD_CONSTANT_TYPE(8577346029146923773);
    xy[1] ^= WORD_CONSTANT_TYPE(17808705383533708044);
}
```

LISTING 6.12: Generated C code to compute the vector addition using inlined bit-packing

While this strategy has mixed results compared to the default implementation, and does not improve on the inlined or the bit-packing strategy, in terms of disk space savings, we believe this strategy can still have a positive impact on execution time. Compared to the inlined strategy, this method has the advantage of the state vector xy being bit-packed. This simplifies the modular addition and vector addition functions. Compared to the bit-packing method, we might expect a performance improvement as a result of the loop unrolling. Experimental results on inlined bit-packed code generation are discussed in Section 6.4.

6.3.5 SIMD code generation

In the previous code generation strategies, we addressed inefficiencies directly introduced by the white-box protection of a SPECK implementation. However, our final code generation strategy is more generic and could potentially be combined with any of the previous strategies. We extend the bit-packed code generation from Section 6.3.3 with instructions from the Advanced Vector Extensions (AVX) and Advanced Vector Extensions 2 (AVX2) instruction sets. For the sake of simplicity, we do not consider $n = 24$ and $n = 32$ in this section.

Single instruction, multiple data (SIMD) allows algorithms to operate on multiple pieces of data, called vectors, at the same time. For example, 16 16-bit integers could be combined into a 256-bit SIMD vector, which could then be manipulated using SIMD instructions. In this section, we will always work with the 256-bit SIMD vector provided by AVX. We also define m to be the amount of packed n-bit integers in a 256-bit SIMD vector. For $n = 16$, $n = 32$, and $n = 64$, this value is equal to 16, 8, and 4, respectively.

SIMD instructions are particularly useful to optimize algorithms where the results can be calculated in parallel. Clearly, this is the case for the matrix-vector product, where an entry i in the result vector `res` is computed as follows:

$$\texttt{res}_i = p((\overline{\mathbf{M}^{(\mathbf{r})}}_{i,x} \;\&\; \mathbf{x}) \oplus (\overline{\mathbf{M}^{(\mathbf{r})}}_{i,y} \;\&\; \mathbf{y}))$$

To fully parallelize this definition, we would require SIMD instructions for three operations: bitwise AND, bitwise XOR, and the parity function p. Unfortunately, there is no such instruction for the parity function. A possible solution to this problem is to parallelly compute an intermediate result, $\texttt{inter}_i$:

$$\texttt{inter}_i = (\overline{\mathbf{M}^{(\mathbf{r})}}_{i,x} \;\&\; \mathbf{x}) \oplus (\overline{\mathbf{M}^{(\mathbf{r})}}_{i,y} \;\&\; \mathbf{y})$$

Function	Description
`_mm256_set1_epi16`	Broadcast a 16-bit integer to the entire 256-bit vector
`_mm256_set1_epi32`	Broadcast a 32-bit integer to the entire 256-bit vector
`_mm256_set1_epi64x`	Broadcast a 64-bit integer to the entire 256-bit vector
`_mm256_and_si256`	Compute the bitwise AND of two 256-bit vectors
`_mm256_xor_si256`	Compute the bitwise XOR of two 256-bit vectors

Table 6.7: SIMD intrinsic functions used in the matrix-vector product

After the parallel computation, the individual intermediate results can be extracted from the SIMD vector. The final step is then to execute the parity function p on each $\texttt{inter}_i$ to obtain $\texttt{res}_i$. Although this solution is not optimal, it still allows us to partially exploit data parallelism for the matrix-vector product using SIMD. To implement this solution, we require 5 SIMD intrinsic functions from the standard library, listed in Table 6.7 [51].

Because these SIMD functions will take SIMD vectors as parameters, we also have to change how $\overline{\mathbf{M}^{(\mathbf{r})}}$ is stored in the C source code. Instead of storing each row individually, m rows of bit-packed integers should be stored together in two SIMD vectors: the x part and the y part:

$$\overline{\mathbf{M}^{(\mathbf{r})}} = \left[\begin{array}{c|c} \overline{\mathbf{M}^{(\mathbf{r})}}_{1,x} & \overline{\mathbf{M}^{(\mathbf{r})}}_{1,y} \\ \vdots & \vdots \\ \overline{\mathbf{M}^{(\mathbf{r})}}_{m,x} & \overline{\mathbf{M}^{(\mathbf{r})}}_{m,y} \\ \hline \overline{\mathbf{M}^{(\mathbf{r})}}_{m+1,x} & \overline{\mathbf{M}^{(\mathbf{r})}}_{m+1,y} \\ \vdots & \vdots \\ \overline{\mathbf{M}^{(\mathbf{r})}}_{2m,x} & \overline{\mathbf{M}^{(\mathbf{r})}}_{2m,y} \\ \hline \vdots & \vdots \\ \hline \overline{\mathbf{M}^{(\mathbf{r})}}_{2n-m+1,x} & \overline{\mathbf{M}^{(\mathbf{r})}}_{2n-m+1,y} \\ \vdots & \vdots \\ \overline{\mathbf{M}^{(\mathbf{r})}}_{2n,x} & \overline{\mathbf{M}^{(\mathbf{r})}}_{m,y} \end{array}\right]$$

Unfortunately, there is no straightforward way to store $\overline{\mathbf{M}^{(\mathbf{r})}}$ as a two-dimensional array of constant SIMD vectors in C source code. As an alternative, we could store the bit-packed integers of $\overline{\mathbf{M}^{(\mathbf{r})}}$ and construct the SIMD matrix when the program starts, however this might result in some runtime overhead. Instead, we utilize technique known as *type punning*, and create a `union` called `simd_union` (Listing 6.13). This `union` allows us to freely convert between an array of m bit-packed integers and the SIMD vector. As a result, creating a constant SIMD vector is equivalent to storing the array of m bit-packed integers, and a bit-packed integer can easily be extracted from the SIMD vector using array indexing.

We can now introduce the SIMD implementation of the matrix-vector product. In Listing 6.14, this implementation is illustrated for $n = 64$ ($m = 4$). The function first constructs two SIMD vectors consisting of copies of the bit-packed input vectors $\mathbf{x}$ and

```
typedef union simd_union {
    WORD_TYPE words[SIMD_PACKED_COUNT];
    SIMD_TYPE simd;
} simd_union;
```

Listing 6.13: Type punning using `union`

```
void matrix_vector_product(simd_union matrix[BLOCK_SIZE /
    ↪ SIMD_PACKED_COUNT][2], WORD_TYPE xy[2], WORD_TYPE res[2]) {
    SIMD_TYPE xy0 = SIMD_SET1(xy[0]);
    SIMD_TYPE xy1 = SIMD_SET1(xy[1]);
    for (size_t i = WORD_SIZE / SIMD_PACKED_COUNT; i--> 0;) {
        simd_union inter0 = {.simd = SIMD_XOR(SIMD_AND(matrix[i
            ↪ ][0].simd, xy0), SIMD_AND(matrix[i][1].simd, xy1))};
        res[0] = (res[0] << 1) | ((WORD_TYPE) WORD_PARITY_FUNCTION
            ↪ (inter0.words[3]));
        res[0] = (res[0] << 1) | ((WORD_TYPE) WORD_PARITY_FUNCTION
            ↪ (inter0.words[2]));
        res[0] = (res[0] << 1) | ((WORD_TYPE) WORD_PARITY_FUNCTION
            ↪ (inter0.words[1]));
        res[0] = (res[0] << 1) | ((WORD_TYPE) WORD_PARITY_FUNCTION
            ↪ (inter0.words[0]));
        simd_union inter1 = {.simd = SIMD_XOR(SIMD_AND(matrix[(
            ↪ WORD_SIZE / SIMD_PACKED_COUNT) + i][0].simd, xy0),
            ↪ SIMD_AND(matrix[(WORD_SIZE / SIMD_PACKED_COUNT) + i
            ↪ ][1].simd, xy1))};
        res[1] = (res[1] << 1) | ((WORD_TYPE) WORD_PARITY_FUNCTION
            ↪ (inter1.words[3]));
        res[1] = (res[1] << 1) | ((WORD_TYPE) WORD_PARITY_FUNCTION
            ↪ (inter1.words[2]));
        res[1] = (res[1] << 1) | ((WORD_TYPE) WORD_PARITY_FUNCTION
            ↪ (inter1.words[1]));
        res[1] = (res[1] << 1) | ((WORD_TYPE) WORD_PARITY_FUNCTION
            ↪ (inter1.words[0]));
    }
}
```

Listing 6.14: Generated C code to compute the matrix-vector product using SIMD functions ($n = 64$)

$\mathbf{y}$. After this, `vector0` and `vector1` are used to parallelly compute the intermediate result $(\overline{\mathbf{M}^{(\mathbf{r})}}_{j,x} \,\&\, \mathbf{x}) \oplus (\overline{\mathbf{M}^{(\mathbf{r})}}_{j,y} \,\&\, \mathbf{y})$ for $i \leq j \leq i+m$, using SIMD bitwise AND and bitwise XOR instructions. The individual intermediate results can then be extracted from the resulting SIMD vector, and the parity function p is applied to obtain m entries for the x part of the output vector. This process is repeated in a similar manner to compute m entries for the y part of the output vector.

In general, $2m$ entries in the result vector are computed in one iteration of the `for` loop, reducing the required number of iterations to $\frac{n}{m}$. As a result, for $n = 16$,

only a single iteration of the for loop is performed. For $n = 32$ and $n = 64$, the number of iterations is 4 and 16 respectively. This partial unrolling might have a similar positive effect on execution time as the inlining from Section 6.3.2. However, potential overhead from the SIMD operations might limit the efficiency gains.

6.4 Comparison

To provide a comprehensive comparison of the Speck encryption performance for the unprotected and protected implementations, we tested three different variants: Speck32/64, Speck64/128, and Speck128/256. We did not test the block sizes 48 and 96, as these parameters are not supported by all code generation strategies. For every variant, we used the keys from the original Speck test vectors to perform the encryptions [6]. However, the choice and length of key should not have an impact on the performance of the protected implementations. Furthermore, to ensure a fair comparison, the same self-equivalence encodings were used when generating C code using different strategies. For simplicity reasons, no external encodings were applied to the generated output. Once again, this decision has a negligible impact on the performance of the white-box Speck implementations.

As mentioned before, we use the GNU Compiler Collection (GCC), version 10.2.0, to compile the C code to executable files. Each Speck implementation was compiled using the following compiler flags: `-Ofast`, `-march=native`, `-pipe`, and `-s`. After compilation, disk space usage was measured using the `du -b` command. Finally, the `perf stat` command was used to measure the execution time of the compiled program. These programs were executed using a single core on a laptop with an `AMD Ryzen 7 PRO 3700U` CPU, running Linux 5.10.14.

In total, this process of generating white-box Speck implementations, compiling the C code, and benchmarking the results, was iterated 100 times for every variant to account for variability in disk space usage and execution times. Although the disk space usage only varies for the sparse matrix, inlined, and inlined bit-packed code generation strategies, we will compare the average of these iterations for all implementations.

6.4.1 Speck32/64

Of all variants we analyzed, Speck32/64 is the most even in terms of disk space usage (Figure 6.10). The unprotected reference implementation takes up $14,432$ bytes of disk space, with the most efficient protected implementations using $17,496$ bytes of disk space. A surprising result in this data is the exact tie between the disk space usage of bit-packed and SIMD implementations. The code structure of these implementations differs substantially, so we assume this tie is a coincidence.

The graph also clearly shows the impact of using the sparse matrix representation, being the third-most efficient code generation strategy in terms of disk space usage. As conjectured in Section 6.3.4, the inlined bit-packed code takes up more disk space than the default implementation. In environments where disk space is tightly constrained,

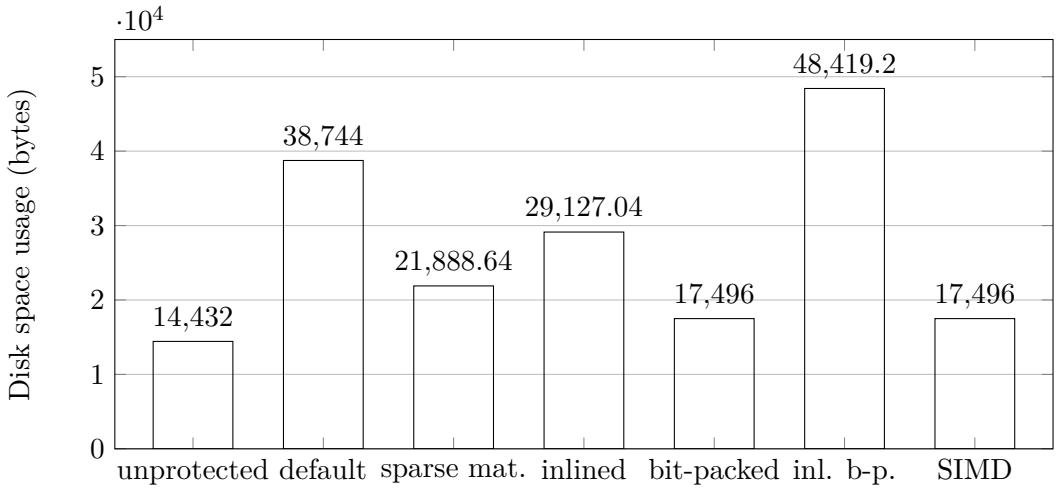

FIGURE 6.10: Average disk space used by different SPECK32/64 implementations

the comparatively simple bit-packed code generation strategy is presumably preferable to more complex SIMD code.

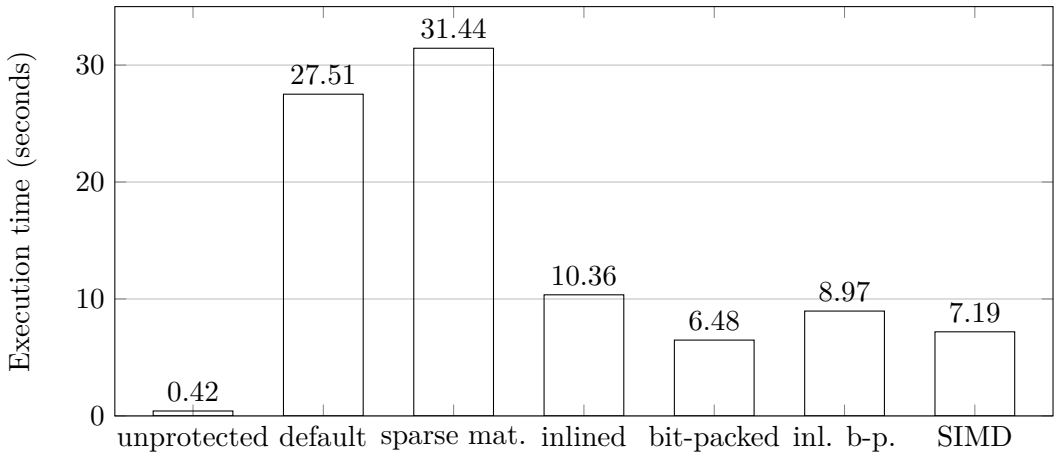

FIGURE 6.11: Average execution time for different SPECK32/64 implementations

In the SPECK32/64 implementations, $10,000,000$ random encryptions were performed upon execution of the program (Figure 6.11). The unprotected implementation finished this in 0.42 seconds, on average, reaching an encryption speed of 761.90 megabytes per second. The most efficient protected implementation is considerably slower, taking on average 6.48 seconds, which results in an encryption speed of only 49.38 megabytes per second. For this block size, the sparse matrix representation seems to have a negative impact on the execution time, which might be explained by the more unpredictable array accesses. Clearly, when encryption speed is the most important factor, bit-packing is also the best code generation

strategy for Speck32/64.

6.4.2 Speck64/128

Because unprotected implementations do not store matrices and vectors which depend on the block size, the required disk space for the unprotected Speck64/128 implementation stays the same. Once again, the code generated using the bit-packed and SIMD strategies is the most efficient in terms of disk space usage, taking about double the disk space of an unprotected implementation. However, the sparse matrix strategy is a close second best option for this block size. This is also the first block size we tested where code generated using the inlined bit-packed strategy requires less disk space than the default protected implementation. Full results can be found in Figure 6.12.

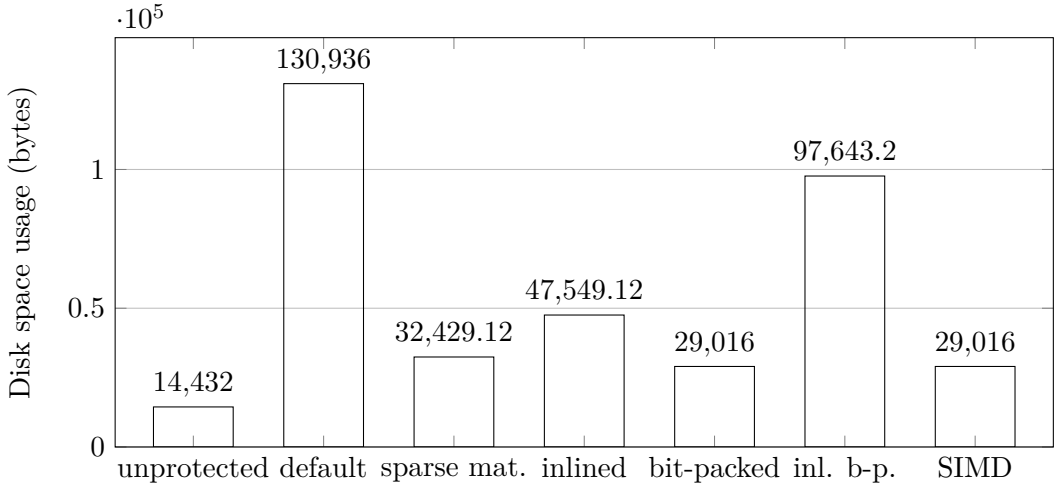

Figure 6.12: Average disk space used by different Speck64/128 implementations

For a block size of 64, $3,000,000$ encryptions were executed, which results in an average execution time of 0.1 seconds for the unprotected implementation. This is equivalent to an encryption speed of 1920 megabytes per second. Still, the bit-packed strategy produces the fastest code (3.8 seconds, 50.53 megabytes per second), however SIMD code is only slightly behind (3.87 seconds). Although the sparse matrix code generation strategy has a large positive impact on disk space usage, it is still less efficient than the default protected implementation in terms of execution time.

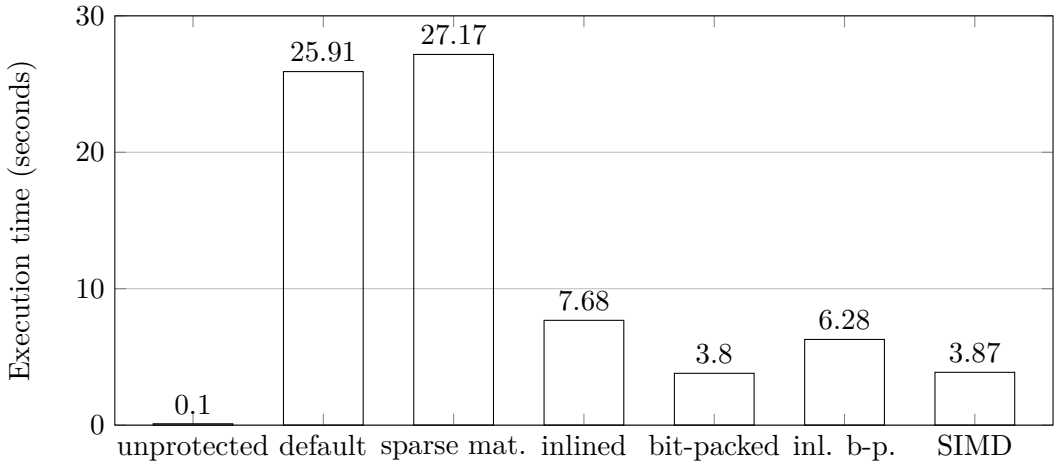

FIGURE 6.13: Average execution time for different SPECK64/128 implementations

6.4.3 Speck128/256

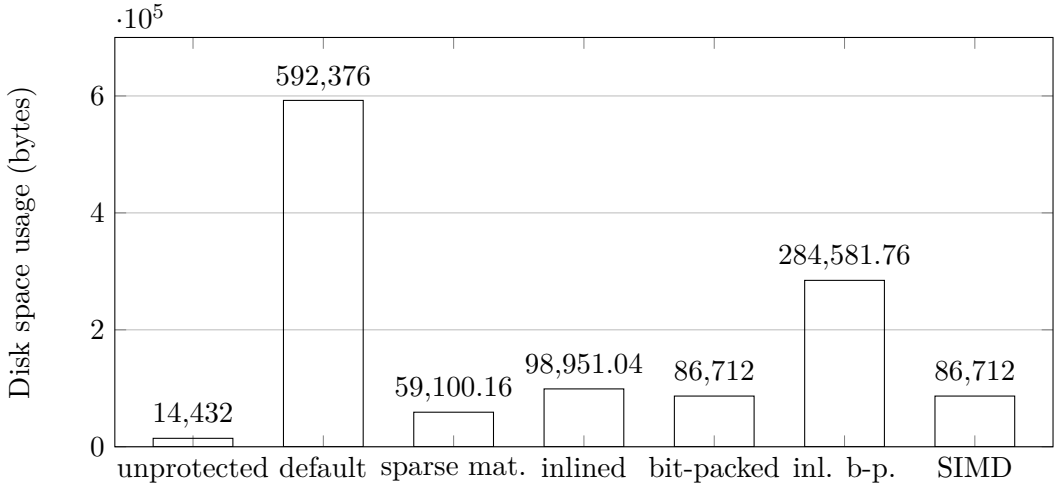

FIGURE 6.14: Average disk space used by different SPECK128/256 implementations

The final disk space usage graph (Figure 6.14) shows us that for block size 128, the sparse matrix representation requires less disk space on average compared to the bit-packed or SIMD implementations. While code generated using these strategies still takes up four times the amount of disk space of an unprotected implementation, it still improves on the default implementation with a reduction of 90%, as was theoretically predicted in Section 6.3.1. For this block size, the inlined bit-packed strategy also extends its lead on the default implementation, but it does not improve on the inlined or bit-packed code generation strategies.

For SPECK128/256 implementations, the number of random encryption iterations

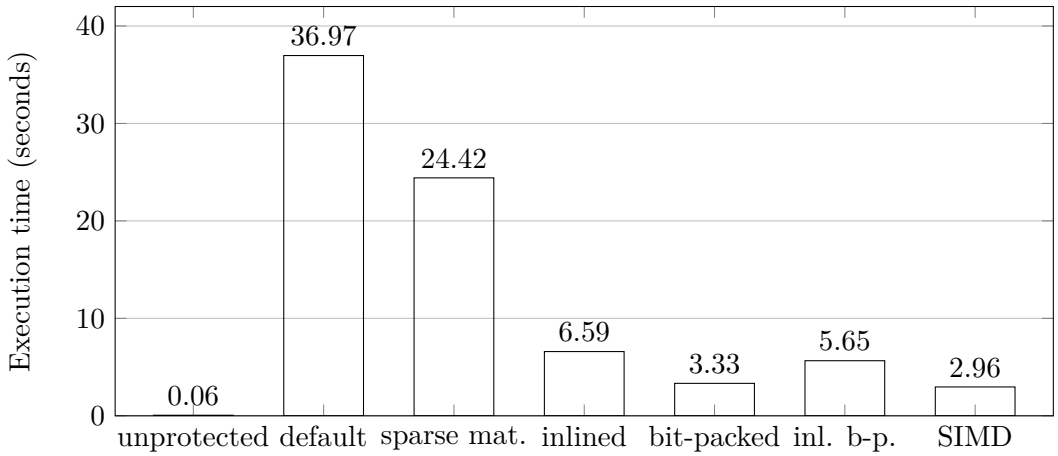

Figure 6.15: Average execution time for different Speck128/256 implementations

was set to $1,000,000$. A graph of the execution times can be found in Figure 6.15). The experimental results show an average encryption speed of 2133.33 megabytes per second for the unprotected implementation. For this block size, the SIMD code is the most performant protected implementation, narrowly beating the bit-packed code generation strategy by 0.37 seconds. The SIMD code reaches an encryption speed of 43.24 megabytes per second, on average. However, if no SIMD instruction set is available on the target platform, then bit-packed code could be a viable alternative, with a slightly lower average encryption speed: 38.43 megabytes per second.

6.5 Conclusion

In this chapter we introduced the Python project we created to generate realistic white-box Speck implementations. This program can generate protected encryption code for any Speck variant. Although the decryption function is currently not supported, all required building blocks are present. In addition to the general architecture of our project, we also described in detail the computation of the protected affine layers, the generation of self-equivalences and external encodings, and the generation of output code.

Furthermore, we discussed five additional strategies to generate protected C code. While some of these strategies only support 32, 64, and 128 bit block sizes, this could easily be modified without too much effort. We showed that for all of these strategies, some performance improvements compared to the default implementation can be expected. Moreover, the techniques introduced in this chapter could also be applied to other mathematical computations relying on the storage of matrices and the computation of a matrix-vector product. In particular, these improvements could be applied to white-box implementations based on self-equivalences of other ARX ciphers.

Finally, we compared an unprotected, reference Speck implementation, a default

white-box Speck implementation, and the code generated by the different code generation strategies for three Speck variants: Speck32/64, Speck64/128, and 128/256. These code generation strategies all improve on the default white-box implementation in varying degrees.

These comparisons also show that the bit-packed code generation strategy provides the most efficient code, both in terms of disk space usage and execution time, for Speck32/64 and Speck64/128. For Speck128/256, the sparse matrix code generation strategy requires the least disk space, and the SIMD code generation strategy results in the fastest encryption speeds. However, the bit-packed code generation strategy is still a close second best in both metrics. Clearly, when all Speck variants are considered, this strategy is the most efficient method to generate white-box Speck code, with a sixfold increase in disk space and a factor 55.5 increase in execution time (Speck128/256).

Chapter 7

Conclusion

White-box cryptography is a young but ambitious field, aiming to protect cryptographic keys against attackers with full access to the implementation. Although white-box cryptography is used extensively in industry, there is currently no public research on white-box implementations of ARX ciphers. To accomplish our first goal, we introduced the first academic method to protect a Speck encryption implementation. For this method, we considered both linear and affine self-equivalence encodings. We showed that these encodings can be applied to Speck rounds without changing the functionality. However, this method is not restricted to Speck. Similar techniques could be used to protect other ARX ciphers, such as Salsa20, ChaCha, or Threefish.

Furthermore, we achieved our second goal by analyzing the security of our method. We discovered that, when only linear self-equivalence encodings are used, only one of the round key bits is adequately masked by the encodings. Moreover, we presented a practical attack to fully recover the linear self-equivalence encodings and external encodings of a protected implementation. This showed that our method is completely insecure when only linear self-equivalences are used. When affine encodings are used, we found that candidate round keys could be computed by guessing the values of only two key bits. However, we were unable to mount an attack to fully recover affine self-equivalence encodings from an encoded round.

Our final goal was to create a functional implementation of our method. We succeeded by creating a Python project to generate white-box Speck code. This implementation can be found in our GitHub repository at https://github.com/jvdsn/white-box-speck. We used this project to calculate the impact of our method on the performance of Speck. Furthermore, we were able to compare five additional strategies to generate protected code, and determined an overall optimal strategy: bit-packed code generation. Lastly, we successfully tested our attack to recover linear self-equivalence encodings from a protected implementation using this project.

Clearly, our method to generate white-box Speck implementations has some serious disadvantages. Most importantly, its security against key recovery attacks is uncertain at best, and fully broken in the worst case, when only linear self-equivalences

are used. On top of this, our method has a considerable impact on the performance of the generated code. In the case of Speck128/256, the most efficient protected implementation was still six times larger and 55.5 times slower than an unprotected implementation.

One possible area for future research is the generation of self-equivalences. In this thesis, we were only able to use two subsets of affine self-equivalences, type 1 and type 2 affine self-equivalences. By composing these self-equivalences, we were able to generate a larger subset of possible encodings. Nevertheless, this subset only covers a fraction of all possible affine self-equivalences. As our implementation framework is modular, it could easily be adapted to utilize a larger subset, or even the full set, of affine self-equivalences. This could result in more secure white-box implementations.

Alternatively, the security of our current method could be analyzed using different approaches in the white-box model. In particular, we did not consider popular techniques based on side-channel analysis, such as differential fault analysis and differential computation analysis. Our Python project could be used to test and compare the efficiency of many different attacks on white-box implementations using self-equivalence encodings.

Finally, there might be other methods to improve the performance of protected Speck implementations. Further research in the storage of binary vectors and matrices, or innovative approaches to speed up the matrix-vector product, could result in better code generation strategies. These strategies could be applied to any white-box implementation using self-equivalence encodings.

Bibliography

[1] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. "White-box cryptography and an AES implementation". In: *International Workshop on Selected Areas in Cryptography*. Springer. 2002, pp. 250–270.

[2] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. "On the (im) possibility of obfuscating programs". In: *Annual international cryptology conference*. Springer. 2001, pp. 1–18.

[3] *Encryption Key Management | Secure Key Box | Intertrust Technologies*. Intertrust Technologies Corporation. URL: https://www.intertrust.com/products/application-protection/secure-key-box/ (visited on Mar. 21, 2021).

[4] Brendan McMillion and Nick Sullivan. "Attacking White-Box AES Constructions". In: *Proceedings of the 2016 ACM Workshop on Software PROtection*. 2016, pp. 85–90.

[5] Adrián Ranea and Bart Preneel. "On Self-Equivalence Encodings in White-Box Implementations". In: *International Workshop on Selected Areas in Cryptography*. Springer. 2020.

[6] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. "The SIMON and SPECK Families of Lightweight Block Ciphers." In: *IACR Cryptol. ePrint Arch.* 2013 (2013), p. 404.

[7] *White Box Cryptography | Essential Encryption | Thales*. Thales. URL: https://cpl.thalesgroup.com/software-monetization/white-box-cryptography (visited on Mar. 21, 2021).

[8] *Whitebox Cryptography - Irdeto*. Irdeto. URL: https://irdeto.com/whitebox-cryptography/ (visited on Mar. 21, 2021).

[9] Joan Daemen and Vincent Rijmen. *The design of Rijndael*. Vol. 2. Springer, 2002.

[10] James A Muir. "A tutorial on white-box AES". In: *Advances in network analysis and its applications* (2012), pp. 209–229.

[11] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. *Advanced Encryption Standard (AES)*. en. Nov. 26, 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[12] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C Van Oorschot. "A white-box DES implementation for DRM applications". In: *ACM Workshop on Digital Rights Management*. Springer. 2002, pp. 1–15.

[13] Matthias Jacob, Dan Boneh, and Edward Felten. "Attacking an obfuscated cipher by injecting faults". In: *ACM workshop on digital rights management*. Springer. 2002, pp. 16–31.

[14] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. "Cryptanalysis of a white box AES implementation". In: *International workshop on selected areas in cryptography*. Springer. 2004, pp. 227–240.

[15] Hamilton E Link and William D Neumann. "Clarifying obfuscation: improving the security of white-box DES". In: *International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume II*. Vol. 1. IEEE. 2005, pp. 679–684.

[16] Yaying Xiao and Xuejia Lai. "A secure implementation of white-box AES". In: *2009 2nd International Conference on Computer Science and its Applications*. IEEE. 2009, pp. 1–6.

[17] Mohamed Karroumi. "Protecting white-box AES with dual ciphers". In: *International conference on information security and cryptology*. Springer. 2010, pp. 278–291.

[18] Jaesung Yoo, Hanjae Jeong, and Dongho Won. "A method for secure and efficient block cipher using white-box cryptography". In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. 2012, pp. 1–8.

[19] Seungkwang Lee, Dooho Choi, and Yong-Je Choi. "Conditional Re-encoding Method for Cryptanalysis-Resistant White-Box AES". In: *ETRI Journal* 37.5 (2015), pp. 1012–1022.

[20] Chung Hun Baek, Jung Hee Cheon, and Hyunsook Hong. "White-box AES implementation revisited". In: *Journal of Communications and Networks* 18.3 (2016), pp. 273–287.

[21] Seungkwang Lee, Taesung Kim, and Yousung Kang. "A masked white-box cryptographic implementation for protecting against differential computation analysis". In: *IEEE Transactions on Information Forensics and Security* 13.10 (2018), pp. 2602–2615.

[22] Julien Bringer, Hervé Chabanne, Emmanuelle Dottax, et al. "White Box Cryptography: Another Attempt." In: *IACR Cryptol. ePrint Arch.* 2006 (2006), p. 468.

[23] Alex Biryukov, Charles Bouillaguet, and Dmitry Khovratovich. "Cryptographic schemes based on the ASASA structure: Black-box, white-box, and public-key". In: *International conference on the theory and application of cryptology and information security*. Springer. 2014, pp. 63–84.

[24] Andrey Bogdanov and Takanori Isobe. "White-box cryptography revisited: Space-hard ciphers". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 1058–1069.

[25] Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. "Cryptanalysis of white-box DES implementations with arbitrary external encodings". In: *International workshop on selected areas in cryptography*. Springer. 2007, pp. 264–277.

[26] Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. "Cryptanalysis of white box DES implementations". In: *International workshop on selected areas in cryptography*. Springer. 2007, pp. 278–295.

[27] Wil Michiels, Paul Gorissen, and Henk DL Hollmann. "Cryptanalysis of a generic class of white-box implementations". In: *International workshop on selected areas in cryptography*. Springer. 2008, pp. 414–428.

[28] Yoni De Mulder, Brecht Wyseur, and Bart Preneel. "Cryptanalysis of a perturbated white-box AES implementation". In: *International Conference on Cryptology in India*. Springer. 2010, pp. 292–310.

[29] Yoni De Mulder, Peter Roelse, and Bart Preneel. "Cryptanalysis of the Xiao–Lai white-box AES implementation". In: *International conference on selected areas in cryptography*. Springer. 2012, pp. 34–49.

[30] Ludo Tolhuizen. "Improved cryptanalysis of an AES implementation". In: *Proceedings of the 33rd WIC Symposium on Information Theory in the Benelux*. 2012, pp. 68–71.

[31] Tancrède Lepoint, Matthieu Rivain, Yoni De Mulder, Peter Roelse, and Bart Preneel. "Two attacks on a white-box AES implementation". In: *International conference on selected areas in cryptography*. Springer. 2013, pp. 265–285.

[32] Brice Minaud, Patrick Derbez, Pierre-Alain Fouque, and Pierre Karpman. "Key-recovery attacks on ASASA". In: *Journal of Cryptology* 31.3 (2018), pp. 845–884.

[33] Patrick Derbez, Pierre-Alain Fouque, Baptiste Lambin, and Brice Minaud. "On recovering affine encodings in white-box implementations". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), pp. 121–149.

[34] Joppe W Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. "Differential computation analysis: Hiding your white-box designs is not enough". In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2016, pp. 215–236.

[35] Estuardo Alpirez Bock, Chris Brzuska, Wil Michiels, and Alexander Treff. "On the ineffectiveness of internal encodings-revisiting the DCA attack on white-box cryptography". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2018, pp. 103–120.

[36] Estuardo Alpirez Bock, Joppe W Bos, Chris Brzuska, Charles Hubain, Wil Michiels, Cristofaro Mune, Eloi Sanfelix Gonzalez, Philippe Teuwen, and Alexander Treff. "White-box cryptography: don't forget about grey-box attacks". In: *Journal of Cryptology* 32.4 (2019), pp. 1095–1143.

[37] Eli Biham and Adi Shamir. "Differential cryptanalysis of the full 16-round DES". In: *Annual International Cryptology Conference*. Springer. 1992, pp. 487–496.

[38] Mitsuru Matsui. "Linear cryptanalysis method for DES cipher". In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1993, pp. 386–397.

[39] Data Encryption Standard et al. "Data encryption standard". In: *Federal Information Processing Standards Publication* (1999), p. 112.

[40] Adrián Ranea. "Self-Equivalences of the Permuted Modular Addition". In: 2021.

[41] Daniel J Bernstein. "The Salsa20 family of stream ciphers". In: *New stream cipher designs*. Springer, 2008, pp. 84–97.

[42] Daniel J Bernstein. "ChaCha, a variant of Salsa20". In: *Workshop Record of SASC*. Vol. 8. 2008, pp. 3–5.

[43] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. "The Skein hash function family". In: *Submission to NIST (round 3)* 7.7.5 (2010), p. 3.

[44] Estuardo Alpirez Bock, Alessandro Amadori, Chris Brzuska, and Wil Michiels. "On the security goals of white-box cryptography". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), pp. 327–357.

[45] Eloi Sanfelix, Cristofaro Mune, and Job de Haas. "Unboxing the white-box". In: *Black Hat EU 2015*. 2015.

[46] Agner Fog. *Instruction tables*. Technical University of Denmark. Oct. 11, 2020. URL: https://www.agner.org/optimize/instruction_tables.pdf (visited on Jan. 16, 2021).

[47] *Welcome to Python.org*. Python Software Foundation. URL: https://www.python.org/ (visited on Feb. 14, 2021).

[48] *SageMath - Open-Source Mathematical Software System*. URL: https://www.sagemath.org/ (visited on Feb. 14, 2021).

[49] *Information technology – Programming languages – C*. Standard. Geneva, CH: International Organization for Standardization, June 2018.

[50] *6.59 Other Built-in Functions Provided by GCC*. Free Software Foundation, Inc. URL: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html (visited on Feb. 7, 2021).

[51] *Intel Intrinsics Guide*. Intel Corporation. Oct. 19, 2020. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (visited on Feb. 13, 2021).