A White-Box Speck Implementation using Self-Equivalence Encodings (Full Version)

Joachim Vandersmissen[1], Adrián Ranea[2], and Bart Preneel[2]

[1] atsec information security
`joachim@atsec.com`
[2] imec-COSIC, KU Leuven, Belgium
`firstname.lastname@esat.kuleuven.be`

Abstract. In 2002, Chow et al. initiated the formal study of white-box cryptography and introduced the CEJO framework. Since then, various white-box designs based on their framework have been proposed, all of them broken. Ranea and Preneel proposed a different method in 2020, called *self-equivalence encodings* and analyzed its security for AES. In this paper, we apply this method to generate the first academic white-box SPECK implementations using self-equivalence encodings. Although we focus on SPECK in this work, our design could easily be adapted to protect other add-rotate-xor (ARX) ciphers. Then, we analyze the security of our implementation against key-recovery attacks. We propose an algebraic attack to fully recover the master key and external encodings from a white-box SPECK implementation, with limited effort required. While this result shows that the linear and affine self-equivalences of SPECK are insecure, we hope that this negative result will spur additional research in higher-degree self-equivalence encodings for white-box cryptography. Finally, we created an open-source Python project implementing our design, publicly available at https://github.com/jvdsn/white-box-speck. We give an overview of five strategies to generate output code, which can be used to improve the performance of the white-box implementation. We compare these strategies and determine how to generate the most performant white-box SPECK code. Furthermore, this project could be employed to test and compare the efficiency of attacks on white-box implementations using self-equivalence encodings.

Keywords: White-box cryptography · Self-equivalence · SPECK.

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 in software, 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. As a result, white-box cryptography is a widely deployed 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 [19]. 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.

On top of this, they introduced the first academic framework (commonly called the CEJO framework) to generate protected implementations in the white-box model, based on the AES block cipher [21]. Shortly after publishing their work on AES, Chow et al. also applied their method to the protection of the DES block cipher [20]. Concurrently, a practical side-channel attack on the white-box DES implementation was published by Jacob et al., using Differential Fault Analysis [27]. However, this attack was not applicable to the AES implementation protected using the CEJO framework. 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 [8].

The publication of these papers sparked more interest in the topic of white-box cryptography, with many new constructions based on DES [31] and AES [40,28,41,29,2] appearing over the years. Unfortunately, all of these implementations have been broken, using both algebraic attacks [39,26,34,22,30,24] and attacks based on side-channel analysis [16,13,12]. All of these designs improved upon or were inspired by the CEJO framework. Consequently, this framework has been analyzed extensively.

On the other hand, the work of Chow et al. also spurred research into entirely different types of constructions, using modified cipher designs [18] or completely new white-box ciphers [9,14,15]. Often this includes different security goals, such as *incompressibility* or *one-wayness* [11]. Some of these new designs have enjoyed limited success, while others were quickly broken [23,35].

In 2016, McMillion et al. used a type of permutations called *self-equivalences* to construct a toy white-box implementation of AES [33]. A self-equivalence of a function is a pair of permutations which can be applied to the start and end of that function without changing the original behavior. McMillion et al. divided AES into substitution and permutation (affine) layers. Then, they computed the self-equivalences of the substitution layers and applied these self-equivalence encodings to the affine layers directly preceding and succeeding the substitution layer. The resulting white-box implementation (also called a *self-equivalence implementation*) is a composition of substitution layers and encoded affine layers containing the round keys. In the same work, they also presented a practical attack to recover the cryptographic key from such implementations.

The work of McMillion et al. received little attention, and was only recently picked up Ranea and Preneel [36]. They 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 from self-equivalence implementations of traditional SPN ciphers, if the S-box does not have differential and linear approximations with probability one. As cryptographically strong S-boxes are designed to resist differential [6] and linear [32] 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.

One possible class of interesting ciphers are add-rotate-xor (ARX) ciphers, whose rounds consist of the three basic operations the name implies: modular addition, bitwise rotation, and bitwise XOR. Because ARX ciphers do not rely on cryptographically strong S-boxes to provide nonlinearity, they are not susceptible to the attack described by Ranea and Preneel. Furthermore, in [37], it was found that the n-bit modular addition has a number of self-equivalences exponential in n. As a result, ciphers employing the modular addition as their only source of nonlinearity are a promising target for research in white-box cryptography based on self-equivalence encodings.

1.1 Contributions

In this paper, we introduce the first academic method to protect SPECK implementations using self-equivalence encodings. Let n be the SPECK word size, and m the number of key words, that is, the key size divided by n [3]. We start by rewriting the SPECK encryption function E_k as a substitution-permutation network (SPN), a composition of affine layers AL and substitution layers SL, with the first and last affine layer having a special structure. To obtain the self-equivalence implementation $\overline{E_k}$, we apply self-equivalence encodings of SL to each of the affine layers. Notably, this design could also be applied to protect other ARX ciphers.

Then, we define the set of linear self-equivalences of SL as $SE_L(SL)$ and the set of affine self-equivalences of SL as $SE_A(SL)$. Using a result from [37], we can determine that $SE_L(SL)$ contains $3 \times 2^{2n+2}$ elements and $SE_A(SL)$ contains $3 \times 2^{2n+8}$ elements. To encode an affine layer, self-equivalences are randomly sampled from $SE_L(SL)$ or $SE_A(SL)$. Provided that n is large enough, it would be impossible for an attacker to brute force the self-equivalence encodings of an encoded affine layer.

However, we found that it is possible to efficiently recover the linear self-equivalence encodings from an encoded affine layer by computing the Gröbner basis of a system of equations. An attacker can then easily compute the round keys, external encodings, and the SPECK master key. Additionally, we also analyzed the security of affine self-equivalence encodings. We show that an attacker can recover the affine self-equivalence encodings from an encoded affine layer up to one free variable. Consequently, the attacker only has to try 2^{m+1} possible configurations to break the white-box SPECK implementation. As m is at most 4 in practice, only $2^5 = 32$ configurations need to be guessed.

We tested these attacks using our Python implementation on consumer hardware. For $n = 64$, the largest SPECK word size available, we found that it took only 16.08 and 42.00 seconds to break the self-equivalence implementations when linear and affine self-equivalence encodings were used, respectively. Unfortunately, we conclude that these self-equivalence encodings are trivially insecure in the white-box model. Still, we hope that our method can be extended using higher-degree self-equivalences in the future to produce a secure white-box SPECK implementation.

We also created a Python implementation of our white-box SPECK method, capable of generating correct white-box SPECK code. This allows us to compare the performance impact of our design to an unprotected SPECK implementation. Because this impact is significant, we extend the program with strategies to generate more performant code. These strategies improve the execution speed of the matrix-vector product, one of the core functions in our implementations, and reduce the disk space required to store the binary matrices and vectors. We believe that these code generation strategies can be of independent interest to improve the performance of 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 self-equivalence implementations of other ARX ciphers.

Finally, we compare an unprotected, reference SPECK implementation, an unoptimized white-box SPECK implementation, and the code generated by the different code generation strategies for three different SPECK variants: SPECK32/64, SPECK64/128, and SPECK128/256. The results show that the bit-packed and SIMD code generation strategies provides the most efficient code, both in terms of disk space usage and execution time. However, these strategies still pale in comparison to the unprotected, reference SPECK implementation, which is 5.4 times smaller and 24.8 times faster than the most efficient white-box implementations.

White-box cryptography is a hard problem, and over the years many white-box designs have been proposed and broken. While many new designs are based on the CEJO framework [20], we attempt to build on the comparatively recent method using self-equivalences [33]. Even though the results show our design is insecure for SPECK, we hope that this work can still be a useful stepping stone in the study of self-equivalence encodings for white-box cryptography.

Outline In Sect. 2, we define some preliminary notation and concepts that will be reused throughout this text. We introduce our approach to apply self-equivalence encodings to SPECK in Sect. 3. Then, in Sect. 4, we will analyze the security of our white-box SPECK implementation using linear and affine self-equivalence encodings. In Sect. 5, we give an overview of our Python project to generate white-box SPECK implementations using self-equivalence encodings and a comparison of five additional strategies to improve the performance of the generated code. Lastly, Sect. 6 contains the conclusions and future work.

2 Preliminaries

In general, lowercase symbols in this paper 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.

An important operation in this paper 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*. Lastly, $x \ggg \alpha$ 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.

2.1 Self-equivalences

We briefly introduce the definition of linear and affine self-equivalences, and its matrix and matrix-vector forms.

Definition 1 (Linear self-equivalence, [10]). *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 2 (Affine self-equivalence, [10]). *Let F be an (n, m)-bit function. Let A be an n-bit linear permutation, a an n-bit constant, B 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 the zero vector.

In this paper, 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.

2.2 Speck

SPECK is a family of lightweight block ciphers proposed by the National Security Agency in 2013 [3]. In particular, SPECK was designed with a focus on performance in software. In this paper, we also use "the SPECK (block) cipher" to refer to the general design of the SPECK family.

The SPECK family consists of ten different instances, depending on the *block size* and *key size* parameters. 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 a SPECK instance with block size 128 and key size 256.

3 Self-equivalences and Speck

This section describes how self-equivalences can be used to create a white-box implementation of SPECK[3]. Being an ARX cipher, the SPECK encryption function is commonly written as a composition of the basic operations: modular addition, bitwise rotation, and bitwise XOR. However, to properly use self-equivalences in our design, SPECK needs to be rewritten as a repeated composition of non-linear and affine layers, similar to a substitution-permutation network (SPN). In the case of SPECK, the non-linear layers will contain the modular addition, and the affine layers contain the bitwise rotation, bitwise XOR, and round keys.

Then, we introduce the definition of an *encoding*: a permutation applied to the start or the end of a function F, to hide the original behavior of F. Encodings can be applied to the round functions of a block cipher to create *encoded implementations*, a type of white-box implementations [19].

We use a special type of encodings, based on self-equivalences of the modular addition, to encode the affine layers of SPECK. The start of the first affine layer and the end of the last affine layer are encoded using random permutations, called *external encodings*. When these encodings are applied to all affine layers

[3] Our method will focus on protecting the SPECK encryption function, but this design could easily be adapted to the SPECK decryption function.

of SPECK, we obtain a *self-equivalence implementation*, a different type of white-box implementations [36]. Note the difference with encoded implementations: in an encoded implementation, entire round functions are encoded; in a self-equivalence implementation, only the affine layers are encoded.

Let us start by rewriting the SPECK encryption function as a substitution-permutation network (SPN). First, we define the encryption function of an SPN.

Definition 3 (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:*

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

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

We can now show that the SPECK encryption function can also be written as a combination of SPN encryption rounds. 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 into 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:

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

This result is also shown visually in Fig. 1. 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.

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:

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

Here, xy contains the bits of x and y in little-endian order, R_α represents a right circular shift of x by α bits, L_β represents a left circular shift of y by β bits, and

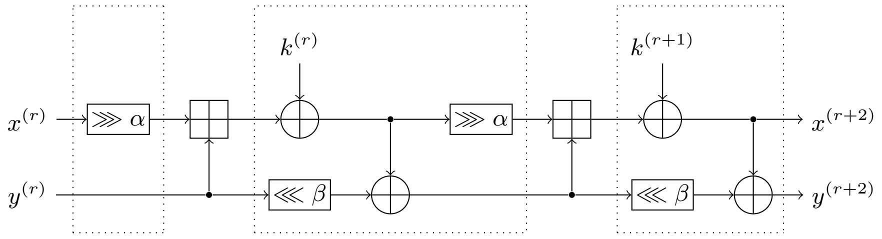

Fig. 1. Diagram of two SPECK encryption rounds, with affine layers indicated using dotted lines.

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.

To protect the key material in $AL^{(r)}$, we need to encode the affine layers. Let us first introduce the definitions of an encoding.

Definition 4 (Encoding, [36]). *Let F be an (n, m)-bit function and let (I, O) be a pair of n-bit and m-bit permutations, respectively. The function $\overline{F} = O \circ F \circ I$ is called an encoded F, and I and O are called the input and output encoding, respectively.*

In our design, the encodings I and O will mainly be self-equivalences of SL when an affine layer is encoded. Therefore, we call these encodings *self-equivalence encodings*. However, the input encoding of the first affine layer and the output encoding of the last affine layer must be random affine permutations, called the external encodings. It is critical to the security of white-box implementations that these external encodings are generated at random and kept secret from the attacker. Without external encodings, our design would be trivially insecure [19]. Now, we define the encoded affine layers.

Definition 5 (Encoded affine layer, [36]). *Let $AL^{(r)}$ be an affine layer of the SPECK cipher, with $1 \leq r \leq n_r$. Then we call $\overline{AL^{(r)}}$ an encoded affine layer, with:*

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

where $((O^{(r)}, o^{(r)}), (I^{(r+1)}, i^{(r+1)}))$ is a self-equivalence of the SPECK substitution layer SL, and $(I^{(1)}, i^{(1)})$ and $(O^{(n_r)}, o^{(n_r)})$ are random affine permutations.

Note that $AL^{(0)}$ will not be encoded: 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. When all affine layers of a SPECK encryption function are encoded using self-equivalence encodings and external encodings, we obtain a self-equivalence implementation of SPECK.

Definition 6 (Self-equivalence implementation, [36]). *Let E_k be the encryption function of the* SPECK *cipher consisting of n_r rounds with word size n. We call $\overline{E_k}$ a self-equivalence implementation of* SPECK*, with:*

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

We can show that a self-equivalence implementation of SPECK, $\overline{E_k}$, is functionally equivalent to E_k, up to the external encodings. Due to the self-equivalence property, the intermediate encodings are canceled:

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

This property is also illustrated in Fig. 2, for two encryption rounds. The dashed lines indicate the substitution layer SL surrounded by its self-equivalence, which can simply be reduced to SL. The encoded affine layers $\overline{AL^{(r)}}$ and $\overline{AL^{(r+1)}}$ are marked by dotted lines.

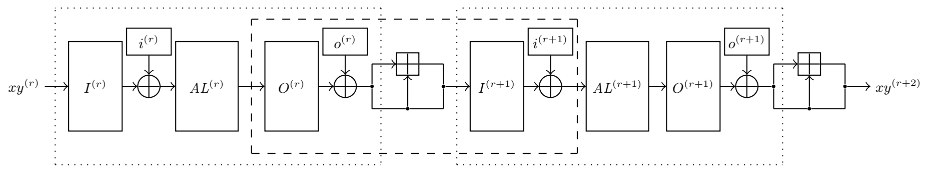

Fig. 2. Diagram of two SPECK SPN encryption rounds encoded using self-equivalences.

3.1 Self-equivalences of *SL*

We use the method described in [37] to generate the self-equivalences of SL. This allows us to randomly sample both linear and affine self-equivalences. For more information, we refer the reader to Section 5.2 of [37].

We call the sets of linear and affine self-equivalences generated using this method $SE_L(SL)$ and $SE_A(SL)$, respectively. When $n > 2$, $|\,SE_L(SL)| = 3 \times 2^{2n+2}$ and $|\,SE_A(SL)| = 3 \times 2^{2n+8}$. 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 3×2^{130} and 3×2^{136} possibilities for linear and affine self-equivalences, respectively, enough to resist a naive brute-force attack. In the next section, we introduce a more extensive security analysis of self-equivalence encodings and show an attacker can still recover self-equivalences without resorting to brute-force.

4 Security analysis

This section analyzes the security of our white-box SPECK design. 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*. A detailed analysis of white-box cryptography security goals is presented by Bock et al. in [11]. In this paper, we focus on the fundamental white-box security feature: resistance to key-recovery attacks.

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 the additional structure of $SE_L(SL)$ and $SE_A(SL)$ 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. Unlike the attack introduced by Ranea and Preneel in [36], which is based on *equivalence problems* and not applicable to SPECK, we analyze self-equivalence equations in bits. 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.

To perform a key-recovery attack on the white-box SPECK implementation, we need to recover the master key k from the self-equivalence implementation $\overline{E_k}$. Unfortunately, the self-equivalence 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. Let n be the SPECK word size, and m the number of key words, that is, the key size divided by n [3]. Suppose $k^{(1)}, \ldots, k^{(m)}$ are known, then compute:

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

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

Note that this approach can be extended to reconstruct k using any sequence of m consecutive round keys, by working backwards to compute the preceding $k^{(r)}$ and $l^{(r)}$ values.

4.1 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 intermediate 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 5) here:

$$\overline{AL^{(r)}} = (\oplus_{o^{(r)}} \circ O^{(r)}) \circ AL^{(r)} \circ (\oplus_{i^{(r)}} \circ I^{(r)}) \tag{1}$$

Because we only consider linear encodings for now, $i^{(r)}$ and $o^{(r)}$ are zero vectors. Consequently, Eq. (1) can be simplified to:

$$\overline{AL^{(r)}} = O^{(r)} \circ AL^{(r)} \circ I^{(r)} \tag{2}$$

This encoded affine layer will be stored as a combination of an encoded matrix $\overline{M^{(r)}}$ and an encoded vector $\overline{v^{(r)}}$:

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

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.

To hide the key material in $v^{(r)}$, $(O^{(r)}, I^{(r+1)})$ need to be randomly generated linear self-equivalences of SL. If the self-equivalences are generated using the method from [37], then $SE_L(SL)$ can be parameterized by a bit vector c of length $2n + 5$, where n is the SPECK word size. We do not describe the full parametrization for $SE_L(SL)$ here, instead, it can be found in the Python project code. For any encoded matrix $\overline{M^{(r)}}$, $c^{(r-1)}$ and $c^{(r)}$ fully define $I^{(r)}$ and $O^{(r)}$, respectively. In other words, if it is possible to recover these bit vectors, an attacker can re-generate $I^{(r)}$ and $O^{(r)}$, peel off the self-equivalence encodings, and compute the round keys and external encodings.

We will now describe a method to recover $c^{(r-1)}$ and $c^{(r)}$ for any intermediate round r. Let X and Y be the matrix forms of the unknown self-equivalence encodings, $I^{(r)}$ and $O^{(r)}$, respectively. Combining this with the definition of $\overline{M^{(r)}}$, we obtain the following equation:

$$\overline{M^{(r)}} = Y M^{(r)} X \tag{3}$$

As $I^{(r)}$ and $O^{(r)}$ are generated using the method from [37], each entry in the matrices X and Y is parameterized by $c^{(r-1)}$ and $c^{(r)}$ respectively. Furthermore, $M^{(r)}$ and $\overline{M^{(r)}}$ are the $2n \times 2n$ matrices known to the attacker. By looking at each entry of these matrices individually, Eq. (3) can be written as a system of $(2n)^2$ equations in $2 \times (2n+5)$ unknowns, the bits in $c^{(r-1)}$ and $c^{(r)}$. Let α_i be the unknowns corresponding to X and β_i the unknowns corresponding to Y. Now, let R be the Boolean polynomial ring in these variables, that is:

$$R = \mathbb{F}_2[\alpha_i, \beta_i]/\langle \alpha_i^2 + \alpha_i, \beta_i^2 + \beta_i \rangle, \text{ for } 1 \le i \le 2n+5 \ .$$

Of course, depending on the density of X, Y, and $M^{(r)}$, many of these equations might not include any α_i or β_i variables. However, by computing the Gröbner basis G of the ideal defined by these equations in R, it is possible to uniquely determine the values of α_i and β_i. We verified this experimentally for every SPECK word size. This in turn reveals the values of $c^{(r-1)}$ and $c^{(r)}$.

An attacker can use this method to recover $c^{(r)}$ for $\overline{M^{(r+1)}}$ and $1 \le r \le m$, where m is the number of SPECK key words. The attacker then re-generates the self-equivalences $(O^{(r)}, I^{(r+1)})$. Because $v^{(r)}$ is always publicly known, the attacker can compute

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

to obtain the round keys $k^{(r)}$ for $r = 1, 2, \ldots, m$.

Similarly, an attacker can recover $c^{(1)}$ from $\overline{M^{(2)}}$ and re-generate the self-equivalence $(O^{(1)}, I^{(2)})$. As with $v^{(r)}$, $M^{(1)}$ is always publicly known, so 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 $I^{(1)}$.

Finally, an attacker can recover $c^{(n_r-1)}$ from $\overline{M^{(n_r-1)}}$ and re-generate the self-equivalence $(O^{(n_r-1)}, I^{(n_r)})$. The attacker then computes

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

to obtain the output external encoding $O^{(n_r)}$.

Note that it is not possible to recover $c^{(1)}$ from $\overline{M^{(1)}}$ or $c^{(n_r-1)}$ from $\overline{M^{(n_r)}}$. Because $M^{(1)}$ and $M^{(n_r)}$ are multiplied by respectively $I^{(1)}$ and $O^{(n_r)}$, random affine permutations, Eq. (3) does not hold.

The most expensive operation in this attack is computing the Gröbner basis for $4n^2$ equations in $2 \times (2n+5)$ variables. Unfortunately, it is notoriously difficult to estimate the time complexity required to compute the Gröbner basis.

Instead, we implemented this attack in Python [1] using SageMath [38] and the PolyBoRi framework [17]. We executed this implementation using a single core on a laptop with an `AMD Ryzen 7 PRO 3700U` CPU, running Linux 5.15.5. The attack took only 16.08 seconds to recover the master key and external encodings from a white-box SPECK128/256 instance. The full results for every SPECK instance can be found in Appendix A.

Clearly, it is feasible to execute this attack using even modest consumer hardware. We conclude that a white-box SPECK implementation using only linear encodings is insecure against key-recovery attacks, even with relatively limited capabilities. 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.

4.2 Security analysis of affine self-equivalences

Knowing that a white-box SPECK implementation using only linear encodings is insecure, we can try to extend this attack to the full design 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 $(O^{(r)}, o^{(r)})$:

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

Once again, we will try to recover the coefficients used to generate a random affine self-equivalence $((O^{(r)}, o^{(r)}), (I^{(r+1)}, i^{(r+1)}))$. In this case, if the self-equivalences are generated using the method from [37], then $SE_A(SL)$ can be parameterized by a bit vector c of length $2n + 11$, where SPECK n is the word size. We previously showed that $c^{(r)}$ can easily be recovered from $\overline{M^{(r)}}$ when only linear encodings are used. However, we found that some coefficients are exclusively used in the constants $i^{(r)}$ and $o^{(r)}$. As a result, we also need to use the definition of $\overline{v^{(r)}}$ to recover the full value of $c^{(r)}$. Furthermore, to simplify our implementation, we will simultaneously recover the bit vector $k^{(r)}$, the round key bits for round r.

Instead of uniquely determining $c^{(r-1)}$, $c^{(r)}$, and $k^{(r)}$ for a round r, we will describe a method to generate possible configurations for these coefficients and key bits. Let (X, x) and (Y, y) be the matrix-vector forms of the unknown self-equivalence encodings, $(I^{(r)}, i^{(r)})$ and $(O^{(r)}, o^{(r)})$, respectively. First, we apply the definition of $\overline{M^{(r)}}$ again to obtain $(2n)^2$ equations, similar to the first step in the attack on linear self-equivalences (Eq. (3)). Let α_i be the unknowns corresponding to (X, x), β_i the unknowns corresponding to (Y, y), and R_1 the Boolean polynomial ring in these variables, that is:

$$R_1 = \mathbb{F}_2[\alpha_i, \beta_i]/\langle \alpha_i^2 + \alpha_i, \beta_i^2 + \beta_i \rangle, \text{ for } 1 \leq i \leq 2n + 11 \enspace .$$

Computing the Gröbner basis G_1 of the ideal defined by these equations in R_1 reveals the values of $4n + 15$ unknowns, slightly less than the total number, $2 \times (2n + 11)$. Now, let z represent the vector $v^{(r)}$, unknown to the attacker. Combining this with (X, x) and (Y, y) and the definition of $\overline{v^{(r)}}$, we obtain the following equation:

$$\overline{v^{(r)}} = Y(z \oplus M^{(r)}x) \oplus y \tag{4}$$

In this case, x is parametrized by by $c^{(r-1)}$, whereas Y and y are parametrized by $c^{(r)}$. Furthermore, recall that $v^{(r)}$ contains $k^{(r)}$, the n unknown round key bits. As $M^{(r)}$ and $\overline{v^{(r)}}$ are known to the attacker, Eq. (4) can be written as a system of $2n$ equations in $2 \times (2n + 11) + n$ unknowns. As before, let α_i be the unknowns corresponding to (X, x) and β_i the unknowns corresponding to (Y, y). We now also introduce γ_j to denote the unknowns corresponding to z. Let R_2 the Boolean polynomial ring in these variables, that is:

$$R_2 = R_1[\gamma_j]/\langle\gamma_j^2 + \gamma_j\rangle, \text{ for } 1 \leq j \leq n \ .$$

However, because the values of $4n+15$ unknowns were revealed by G_1, we also define the quotient ring $Q = R_2/G_1$. Finally, we again compute the Gröbner basis G_2 of the ideal defined by the equations of $\overline{v^{(r)}}$ in Q. This uniquely determines the values of all but one of the unknowns, resulting in two possible configurations for $c^{(r-1)}$, $c^{(r)}$, and $k^{(r)}$.

An attacker can then follow the same process described in Sect. 4.1 to recover the possible round keys and external encodings $I^{(1)}$ and $O^{(n_r)}$. In total, 2^{m+1} possible configurations must be enumerated, with m the number of key words. Because m is at most 4 for SPECK, this exponential function is no problem in practice. We implemented this attack in Python [1] using SageMath [38] and PolyBoRi [17] and executed it using the same setup used for linear encodings. Now the attack took 42.00 seconds to recover the master key and external encodings from a white-box SPECK128/256 instance. Again, the full results can be found in Appendix A.

Although this attack is certainly more expensive than the one for linear encodings, the master key and external encodings are still easily extracted in practice. Consequently, we must conclude that our white-box SPECK method is insecure in the white-box model. However, a higher level of security might be achieved by using quadratic, cubic, and even quartic self-equivalences.

5 Implementation

In previous sections, 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 project is publicly available in our GitHub repository[4]. Our program

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

to generate white-box SPECK implementations[5] is written in Python, a free and open source programming language [1]. We chose Python because its 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 [38]. SageMath is a free and open source mathematics package, which is used extensively for mathematical computations throughout the project.

Our program generates white-box SPECK implementations in four major steps. First, the program takes the block size $2n$ and master key k as input. Using the SPECK key schedule, k is transformed into round keys $k^{(r)}$, and $M^{(r)}$ and $v^{(r)}$ (see Sect. 4.1) are computed. Then, for each round r, random self-equivalence encodings are generated to encode $M^{(r)}$ and $v^{(r)}$. In this step, the random external encodings are also generated and applied to round 1 and n_r. Finally, using $\overline{M^{(r)}}$ and $\overline{v^{(r)}}$, the program generates the output code, a white-box SPECK implementation and its inverse external encodings.

Currently, this output code is exclusively 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. However, our project could easily be adapted to return source or compiled code for other programming languages.

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

The encryption function 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. Conversion to and from binary is done big-endian. The other three functions use a standard textbook implementation. For example, the modular addition simply performs the addition with carry algorithm on each individual bit, ignoring the final carry to perform the modular reduction. In the case of the matrix-vector product, two `for` loops are used to compute the resulting vector. For the vector addition, the generated code performs an XOR operation for each bit in the vector. We call this the default code generation strategy; in the next section we will consider techniques to implement these functions more efficiently.

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

[5] Our project currently only supports the generation of white-box SPECK encryption code. However, the existing project could easily be modified to also generate white-box SPECK decryption implementations. When discussing the generated code in this section, we always refer to SPECK encryption.

a 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, $n_r + 1$ two-dimensional arrays and $n_r + 1$ one-dimensional arrays are generated by the default code generation strategy.

5.1 Code generation strategies

Although the method described previously generates correct and functional C code, this code is far from optimal. We introduce five additional code generation strategies to improve the efficiency of the generated C code.

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 call this the *sparse matrix representation.* In addition to reducing the disk space used by the generated C code, using the sparse matrix representation also simplifies the matrix-vector product. Similar to the sparse matrix representation, we can also use a *sparse vector representation* for the vectors $\overline{v^{(r)}}$. The vector addition can also be modified to take advantage of the sparse vector representation.

Inlined code generation Before the C code is generated, the contents of $\overline{M^{(r)}}$ and $\overline{v^{(r)}}$ are already known. Therefore, it is possible to generate $n_r + 1$ 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.

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 considerably 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.

Inlined bit-packed code generation This code generation strategy combines the previous two strategies. n bits are bit-packed in an n-bit unsigned integer, and used in $n_r + 1$ different functions for the matrix-vector product and for the vector addition. Compared to the inlined strategy, this method has the advantage of the state vector xy being bit-packed. Compared to the bit-packing method, we might expect a performance improvement as a result of the loop unrolling in the inlined functions.

SIMD code generation We extend the bit-packed code generation with instructions from the Advanced Vector Extensions (AVX) and Advanced Vector Extensions 2 (AVX2) instruction sets. Single instruction, multiple data

(SIMD) allows algorithms to operate on multiple pieces of data, called vectors, at the same time. For example, sixteen 16-bit integers could be combined into a 256-bit SIMD vector, which could then be manipulated using SIMD instructions. For the sake of simplicity, our implementation does not consider $n = 24$ and $n = 32$.

5.2 Comparison

To provide a comprehensive comparison of the SPECK encryption performance for the unprotected and self-equivalence 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 [3]. However, the choice and length of key should not have an impact on the performance of the self-equivalence implementations. Furthermore, to ensure a fair comparison, the same affine self-equivalence encodings were used when generating C code using different strategies. We give an overview of the results for each of the three variants, the full details of the experiments can be found in Appendix B.

In the case of SPECK32/64, the unprotected reference implementation takes up 16 320 bytes of disk space, with the smallest self-equivalence implementation, the bit-packed implementation, using only 19 552 bytes of disk space. To compare the performance, 1 000 000 random encryptions were performed upon execution of the program. On average, the unprotected implementation finished this in 0.22 seconds at 4.0 GHz, reaching a throughput of 220 cycles per byte (c/b). The most efficient self-equivalence implementation, again the bit-packed implementation, is considerably slower, taking on average 2.26 seconds, which results in a throughput of 2260 c/b.

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. The smallest self-equivalence implementations are the SIMD and bit-packed implementations, using 31 072 and 31 080 bytes respectively. For a block size of 64 bits, 300 000 encryptions were executed, which results in an average execution time of 0.08 seconds for the unprotected implementation. This is equivalent to a throughput of 133 c/b. Here, the SIMD strategy also produces the fastest code (1.31 seconds, 2183 c/b), however bit-packed code is only slightly behind (1.51 seconds, 2517 c/b).

Finally, for SPECK128/256 implementations, the sparse matrix representation requires a similar amount of disk space on average (95 345.6 bytes) compared to the bit-packed (88 760 bytes) or SIMD (88 752 bytes) implementations. While code generated using these strategies still takes up six times the amount of disk space of an unprotected implementation, it still improves on the default self-equivalence implementation with a reduction of 85%. For this block size, the number of random encryption iterations was set to 100 000. The experimental results show an average throughput of 125 c/b for the unprotected implementation, while the bit-packed code is the most performant self-equivalence implementation, reaching a throughput of 2825 c/b on average.

6 Conclusion

In this work, we introduced the first academic method to protect white-box SPECK implementations using self-equivalence encodings. We showed that these encodings can be applied to SPECK rounds, ostensibly hiding the round keys in encoded affine layers. Similar techniques could be used to protect other ARX ciphers, such as SALSA20 [5], CHACHA20 [4], or THREEFISH [25]. We also analyzed the security of our design against key-recovery attacks. We presented practical attacks to fully recover the self-equivalence encodings and external encodings of a self-equivalence implementation, showing that our method is completely insecure in the white-box model. Finally, we created a Python project to generate self-equivalence implementations using our method. 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 output C code, and determined an overall optimal strategy: bit-packed code generation.

One possible area for future research is the generation of self-equivalences. In this paper, we only employed linear and affine self-equivalences. Extending this design to quadratic, cubic, or higher-degree self-equivalences could result in more secure white-box implementations. Alternatively, the security of our current method could be analyzed using several approaches in the white-box model. In particular, we did not consider the known techniques based on side-channel analysis, such as differential fault analysis [7] and differential computation analysis [12]. Our Python project could be used to test and compare the efficiency of several attacks on white-box implementations using self-equivalence encodings.

Acknowledgements Joachim Vandersmissen would like to thank atsec information security for its support. Adrián Ranea is supported by a PhD Fellowship from the Research Foundation – Flanders (FWO). The authors would like to thank the anonymous reviewers for their comments and suggestions.

References

1. Welcome to python.org, https://www.python.org/
2. Baek, C.H., Cheon, J.H., Hong, H.: White-box AES implementation revisited. J. Commun. Networks **18**(3), 273–287 (2016)
3. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers. IACR Cryptol. ePrint Arch. p. 404 (2013)
4. Bernstein, D.J.: ChaCha, a variant of Salsa20. In: Workshop Record of SASC. vol. 8, pp. 3–5 (2008)
5. Bernstein, D.J.: The Salsa20 family of stream ciphers. In: The eSTREAM Finalists, Lecture Notes in Computer Science, vol. 4986, pp. 84–97. Springer (2008)

6. Biham, E., Shamir, A.: Differential cryptanalysis of the full 16-round DES. In: CRYPTO. Lecture Notes in Computer Science, vol. 740, pp. 487–496. Springer (1992)
7. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: CRYPTO. Lecture Notes in Computer Science, vol. 1294, pp. 513–525. Springer (1997)
8. Billet, O., Gilbert, H., Ech-Chatbi, C.: Cryptanalysis of a white box AES implementation. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 3357, pp. 227–240. Springer (2004)
9. Biryukov, A., Bouillaguet, C., Khovratovich, D.: Cryptographic schemes based on the ASASA structure: Black-box, white-box, and public-key (extended abstract). In: ASIACRYPT (1). Lecture Notes in Computer Science, vol. 8873, pp. 63–84. Springer (2014)
10. Biryukov, A., Cannière, C.D., Braeken, A., Preneel, B.: A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 2656, pp. 33–50. Springer (2003)
11. Bock, E.A., Amadori, A., Brzuska, C., Michiels, W.: On the security goals of white-box cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(2), 327–357 (2020)
12. Bock, E.A., Bos, J.W., Brzuska, C., Hubain, C., Michiels, W., Mune, C., Gonzalez, E.S., Teuwen, P., Treff, A.: White-box cryptography: don't forget about grey-box attacks. Journal of Cryptology **32**(4), 1095–1143 (2019)
13. Bock, E.A., Brzuska, C., Michiels, W., Treff, A.: On the ineffectiveness of internal encodings - revisiting the DCA attack on white-box cryptography. In: ACNS. Lecture Notes in Computer Science, vol. 10892, pp. 103–120. Springer (2018)
14. Bogdanov, A., Isobe, T.: White-box cryptography revisited: Space-hard ciphers. In: CCS. pp. 1058–1069. ACM (2015)
15. Bogdanov, A., Isobe, T., Tischhauser, E.: Towards practical whitebox cryptography: Optimizing efficiency and space hardness. In: ASIACRYPT (1). Lecture Notes in Computer Science, vol. 10031, pp. 126–158 (2016)
16. Bos, J.W., Hubain, C., Michiels, W., Teuwen, P.: Differential computation analysis: Hiding your white-box designs is not enough. In: CHES. Lecture Notes in Computer Science, vol. 9813, pp. 215–236. Springer (2016)
17. Brickenstein, M., Dreyer, A.: Polybori: A framework for Gröbner-basis computations with boolean polynomials. J. Symb. Comput. **44**(9), 1326–1345 (2009)
18. Bringer, J., Chabanne, H., Dottax, E.: White box cryptography: Another attempt. IACR Cryptol. ePrint Arch. p. 468 (2006)
19. Chow, S., Eisen, P.A., Johnson, H., van Oorschot, P.C.: White-box cryptography and an AES implementation. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 2595, pp. 250–270. Springer (2002)
20. Chow, S., Eisen, P.A., Johnson, H., van Oorschot, P.C.: A white-box DES implementation for DRM applications. In: Digital Rights Management Workshop. Lecture Notes in Computer Science, vol. 2696, pp. 1–15. Springer (2002)
21. Daemen, J., Rijmen, V.: The Design of Rijndael - The Advanced Encryption Standard (AES), Second Edition. Information Security and Cryptography, Springer (2020)
22. De Mulder, Y., Roelse, P., Preneel, B.: Cryptanalysis of the Xiao - Lai white-box AES implementation. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 7707, pp. 34–49. Springer (2012)

23. De Mulder, Y., Wyseur, B., Preneel, B.: Cryptanalysis of a perturbated white-box AES implementation. In: INDOCRYPT. Lecture Notes in Computer Science, vol. 6498, pp. 292–310. Springer (2010)
24. Derbez, P., Fouque, P., Lambin, B., Minaud, B.: On recovering affine encodings in white-box implementations. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 121–149 (2018)
25. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein hash function family. Submission to NIST (round 3) **7**(7.5), 3 (2010)
26. Goubin, L., Masereel, J., Quisquater, M.: Cryptanalysis of white box DES implementations. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 4876, pp. 278–295. Springer (2007)
27. Jacob, M., Boneh, D., Felten, E.W.: Attacking an obfuscated cipher by injecting faults. In: Digital Rights Management Workshop. Lecture Notes in Computer Science, vol. 2696, pp. 16–31. Springer (2002)
28. Karroumi, M.: Protecting white-box AES with dual ciphers. In: ICISC. Lecture Notes in Computer Science, vol. 6829, pp. 278–291. Springer (2010)
29. Lee, S., Choi, D., Choi, Y.J.: Conditional re-encoding method for cryptanalysis-resistant white-box AES. ETRI Journal **37**(5), 1012–1022 (2015)
30. Lepoint, T., Rivain, M., De Mulder, Y., Roelse, P., Preneel, B.: Two attacks on a white-box AES implementation. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 8282, pp. 265–285. Springer (2013)
31. Link, H.E., Neumann, W.D.: Clarifying obfuscation: Improving the security of white-box DES. In: ITCC (1). pp. 679–684. IEEE Computer Society (2005)
32. Matsui, M.: Linear cryptanalysis method for DES cipher. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 765, pp. 386–397. Springer (1993)
33. McMillion, B., Sullivan, N.: Attacking white-box AES constructions. In: SPRO@CCS. pp. 85–90. ACM (2016)
34. Michiels, W., Gorissen, P., Hollmann, H.D.L.: Cryptanalysis of a generic class of white-box implementations. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 5381, pp. 414–428. Springer (2008)
35. Minaud, B., Derbez, P., Fouque, P., Karpman, P.: Key-recovery attacks on ASASA. J. Cryptol. **31**(3), 845–884 (2018)
36. Ranea, A., Preneel, B.: On self-equivalence encodings in white-box implementations. In: SAC. Lecture Notes in Computer Science, vol. 12804, pp. 639–669. Springer (2020)
37. Ranea, A., Vandersmissen, J., Preneel, B.: Implicit white-box implementations: White-boxing ARX ciphers. IACR Cryptol. ePrint Arch. (2022)
38. The Sage Developers: SageMath, the Sage Mathematics Software System (Version 9.4) (2021), `https://www.sagemath.org`
39. Wyseur, B., Michiels, W., Gorissen, P., Preneel, B.: Cryptanalysis of white-box DES implementations with arbitrary external encodings. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 4876, pp. 264–277. Springer (2007)
40. Xiao, Y., Lai, X.: A secure implementation of white-box AES. In: 2009 2nd International Conference on Computer Science and its Applications. pp. 1–6. IEEE (2009)
41. Yoo, J., Jeong, H., Won, D.: A method for secure and efficient block cipher using white-box cryptography. In: ICUIMC. pp. 89:1–89:8. ACM (2012)

A Key-recovery attack experimental results

In Sect. 4.1 and Sect. 4.2 we introduced attacks on white-box SPECK implementations using linear, respectively affine, self-equivalences. We executed these attacks using a single core on a laptop with an `AMD Ryzen 7 PRO 3700U` CPU, running Linux 5.15.5. The full results of our experiments in the linear case can be found in Table 1.

Table 1. Time required to recover the master key and external encodings when linear encodings are used.

Word size n	Key words m	Execution time (s)
16	4	1.38
24	3	2.05
24	4	2.43
32	3	3.35
32	4	3.91
48	2	7.85
48	3	7.30
64	2	14.10
64	3	14.57
64	4	16.08

Table 2 contains the full results of our experiments when affine encodings are used.

Table 2. Time required to recover the master key and external encodings when affine encodings are used.

Word size n	Key words m	Execution time (s)
16	4	4.24
24	3	5.75
24	4	7.39
32	3	7.31
32	4	14.62
48	2	13.52
48	3	20.72
64	2	28.77
64	3	39.42
64	4	42.00

B Comparison of code generation strategies

We used the GNU Compiler Collection (GCC), version 11.1.0, 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 5.15.5.

In total, this process of generating white-box SPECK implementations, compiling the C code, and benchmarking the results, was iterated 50 times for every variant to account for variability in disk space usage and execution times. Figures 3 to Fig. 8 show the experimental results for each of three different SPECK variants we considered: SPECK32/64, SPECK64/128, and SPECK128/256.

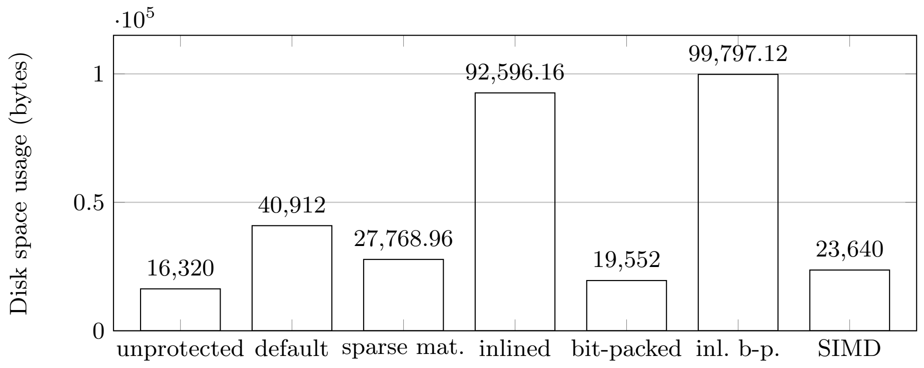

Fig. 3. Average disk space used by different SPECK32/64 implementations.

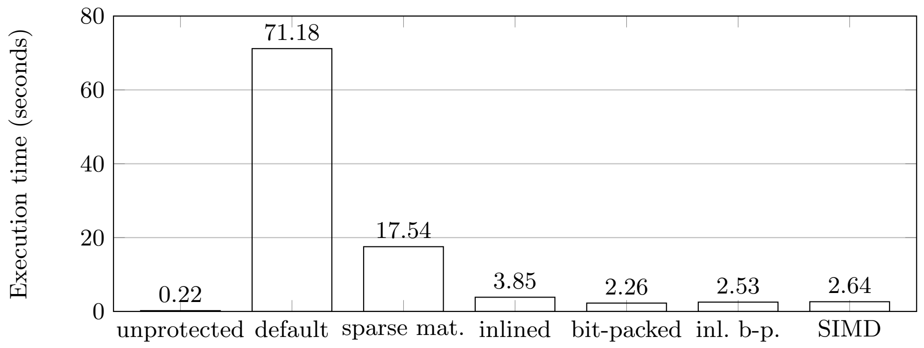

Fig. 4. Average execution time for different SPECK32/64 implementations.

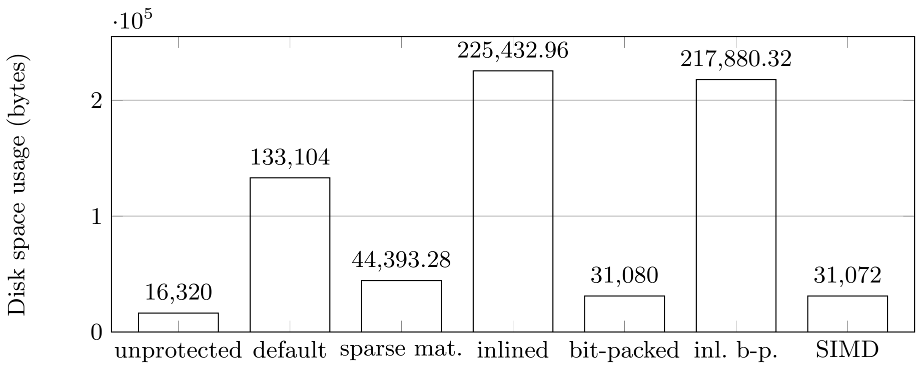

Fig. 5. Average disk space used by different SPECK64/128 implementations.

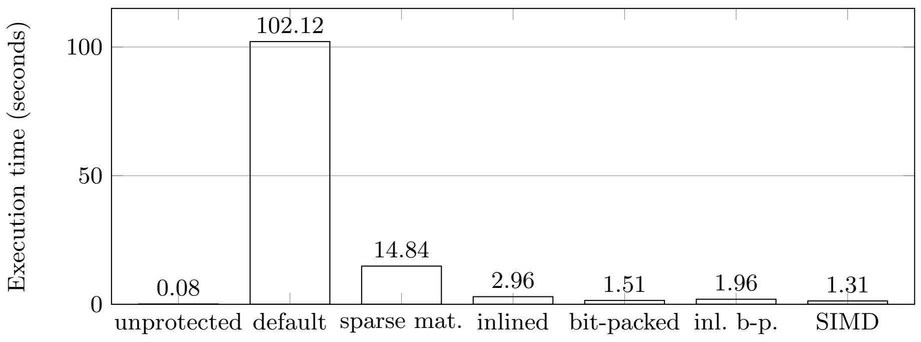

Fig. 6. Average execution time for different SPECK64/128 implementations.

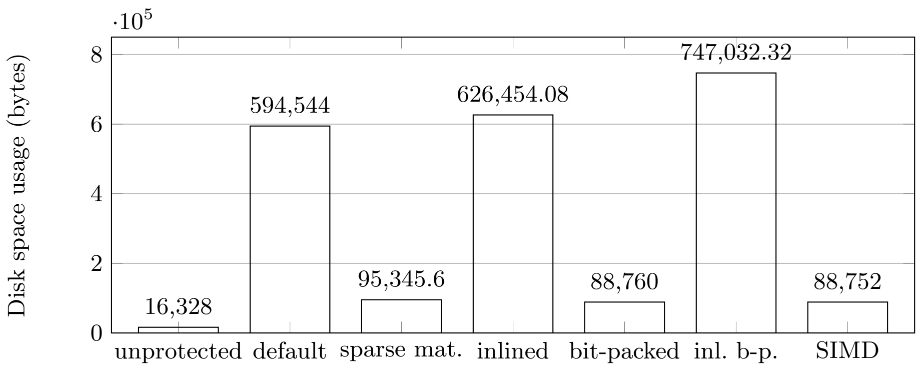

Fig. 7. Average disk space used by different SPECK128/256 implementations.

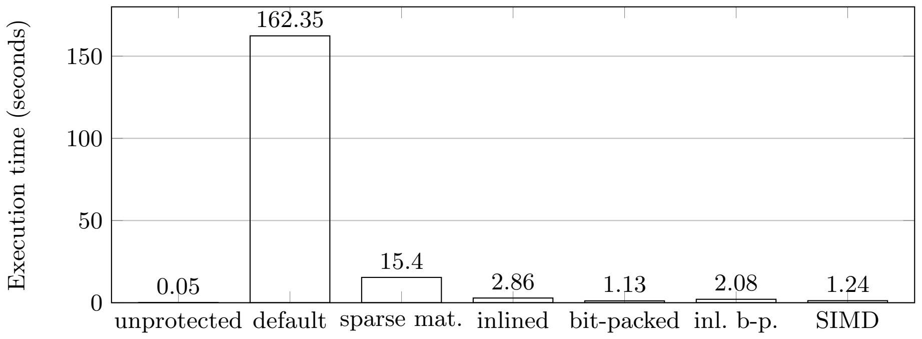

Fig. 8. Average execution time for different SPECK128/256 implementations.